

Distributed Programming with Ice

**Michi Henning
Mark Spruiell**

With contributions by

**Benoit Foucher, Marc Laukien,
Matthew Newhook, Bernard Normier**

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and ZeroC was aware of the trademark claim, the designations have been printed in initial caps or all caps.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Copyright © 2003-2004 by ZeroC, Inc.
mailto:info@zeroc.com
<http://www.zeroc.com>

Revision 1.4.0, 24 May 2004

This revision of the documentation describes Ice version 1.4.0.

The Ice source distribution makes use of a number of third-party products:

- Berkeley DB, developed by Sleepycat Software (<http://www.sleepycat.com>)
- bzip2/libbzip2, developed by Julian R. Seward (<http://sources.redhat.com/bzip2>)
- The OpenSSL Toolkit, developed by the OpenSSL Project (<http://www.openssl.org>)
- SSLeay, developed by Eric Young (mailto:eay@cryptsoft.com)
- Expat, developed by James Clark (<http://www.libexpat.org>)

See the Ice source distribution for the license agreements for each of these products.

Note: The documentation contains a number of unresolved cross-references to material that has not yet been written, but will be added in the future. Such cross-references are marked “XREF”.

Contents

Chapter 1	Introduction	1
1.1	Introduction	1
1.2	The Internet Communications Engine (Ice)	4
1.3	Organization of this Book	4
1.4	Typographical Conventions	5
1.5	Source Code Examples	6
1.6	Contacting the Authors	6
1.7	Ice Support	6
Part I Ice Overview		7
Chapter 2	Ice Overview	9
2.1	Chapter Overview	9
2.2	The Ice Architecture	9
2.3	Ice Services	24
2.4	Architectural Benefits of Ice	26
2.5	A Comparison with CORBA	28
Chapter 3	A Hello World Application	33
3.1	Chapter Overview	33
3.2	Writing a Slice Definition	34
3.3	Writing an Ice Application with C++	34
3.4	Writing an Ice Application with Java	42
3.5	Summary	49

Part II Ice Core Concepts	51
<hr/>	
Chapter 4	The Slice Language
4.1	Chapter Overview
4.2	Introduction
4.3	Compilation
4.4	Source Files
4.5	Lexical Rules
4.6	Basic Slice Types
4.7	User-Defined Types
4.8	Interfaces, Operations, and Exceptions
4.9	Classes
4.10	Forward Declarations
4.11	Modules
4.12	Type IDs
4.13	Operations on Object
4.14	Local Types
4.15	The Ice Module
4.16	Names and Scoping
4.17	Metadata
4.18	Using the Slice Compilers
4.19	A Comparison of Slice and CORBA IDL
4.20	Summary
<hr/>	
Chapter 5	Slice for a Simple File System
5.1	Chapter Overview
5.2	The File System Application
5.3	Slice Definitions for the File System
5.4	The Complete Definition

Chapter 6	Client-Side Slice-to-C++ Mapping	143
6.1	Chapter Overview	143
6.2	Introduction	143
6.3	Mapping for Identifiers	144
6.4	Mapping for Modules	145
6.5	The <code>Ice</code> Namespace	146
6.6	Mapping for Simple Built-In Types	146
6.7	Mapping for User-Defined Types	147
6.8	Mapping for Constants	150
6.9	Mapping for Exceptions	151
6.10	Mapping for Run-Time Exceptions	154
6.11	Mapping for Interfaces	155
6.12	Mapping for Operations	163
6.13	Exception Handling	169
6.14	Mapping for Classes	171
6.15	<code>slice2cpp</code> Command-Line Options	186
6.16	A Comparison with the CORBA C++ Mapping	187
Chapter 7	Developing a File System Client in C++	189
7.1	Chapter Overview	189
7.2	The C++ Client	189
7.3	Summary	194
Chapter 8	Client-Side Slice-to-Java Mapping	197
8.1	Chapter Overview	197
8.2	Introduction	197
8.3	Mapping for Identifiers	198
8.4	Mapping for Modules	199
8.5	The <code>Ice</code> Package	200
8.6	Mapping for Simple Built-In Types	200
8.7	Mapping for User-Defined Types	200
8.8	Mapping for Constants	205
8.9	Mapping for Exceptions	206
8.10	Mapping for Run-Time Exceptions	207
8.11	Mapping for Interfaces	208
8.12	Mapping for Operations	214
8.13	Exception Handling	220
8.14	Mapping for Classes	222
8.15	Packages	226
8.16	<code>slice2java</code> Command-Line Options	227

Chapter 9	Developing a File System Client in Java	229
9.1	Chapter Overview	229
9.2	The Java Client	229
9.3	Summary	233
Chapter 10	Server-Side Slice-to-C++ Mapping	235
10.1	Chapter Overview	235
10.2	Introduction	235
10.3	The Server-Side <code>main</code> Function	236
10.4	Mapping for Interfaces	250
10.5	Parameter Passing	253
10.6	Raising Exceptions	255
10.7	Object Incarnation	256
10.8	Summary	261
Chapter 11	Developing a File System Server in C++	263
11.1	Chapter Overview	263
11.2	Implementing a File System Server	263
11.3	Summary	279
Chapter 12	Server-Side Slice-to-Java Mapping	281
12.1	Chapter Overview	281
12.2	Introduction	281
12.3	The Server-Side <code>main</code> Function	282
12.4	Mapping for Interfaces	287
12.5	Parameter Passing	290
12.6	Raising Exceptions	291
12.7	Tie Classes	292
12.8	Object Incarnation	295
12.9	Summary	299
Chapter 13	Developing a File System Server in Java	301
13.1	Chapter Overview	301
13.2	Implementing a File System Server	301
13.3	Summary	310

Chapter 14	Ice Properties and Configuration	311
14.1	Chapter Overview	311
14.2	Properties	311
14.3	Configuration Files	313
14.4	Setting Properties on the Command Line	314
14.5	The <code>Ice.Config</code> Property	314
14.6	Command-Line Parsing and Initialization	316
14.7	The <code>Ice.ProgramName</code> property	317
14.8	Using Properties Programmatically	317
14.9	Summary	328
Chapter 15	Threads and Concurrency with C++	329
15.1	Chapter Overview	329
15.2	Introduction	329
15.3	The Ice Threading Model	330
15.4	Library Overview	331
15.5	Mutexes	331
15.6	Recursive Mutexes	337
15.7	Read-Write Recursive Mutexes	339
15.8	Timed Locks	343
15.9	Monitors	346
15.10	Threads	356
15.11	Portable Signal Handling	364
15.12	Summary	365

Part III Advanced Ice **367**

Chapter 16	The Ice Run Time in Detail	369
16.1	Introduction	369
16.2	Communicators	370
16.3	Object Adapters	374
16.4	Object Identity	381
16.5	The <code>Ice::Current</code> Object	382
16.6	Servant Locators	383
16.7	Server Implementation Techniques	399
16.8	The <code>Ice::Context</code> Parameter	422
16.9	Invocation Timeouts	428
16.10	Oneway Invocations	430
16.11	Datagram Invocations	434
16.12	Batched Invocations	436
16.13	Testing Proxies for Dispatch Type	438
16.14	The <code>Ice::Logger</code> Interface	438
16.15	The <code>Ice::Stats</code> Interface	440
16.16	Location Transparency	441
16.17	A Comparison of the Ice and CORBA Run Time	442
16.18	Summary	444
Chapter 17	Asynchronous Programming	447
17.1	Chapter Overview	447
17.2	Introduction	447
17.3	Using AMI	450
17.4	Using AMD	458
17.5	Summary	465
Chapter 18	Facets and Versioning	467
18.1	Introduction	467
18.2	Concept and APIs	467
18.3	The Versioning Problem	474
18.4	Versioning with Facets	480
18.5	Facet Selection	481
18.6	Behavioral Versioning	482
18.7	Design Considerations	485
18.8	Summary	486

Chapter 19	The Ice Protocol	487
19.1	Chapter Overview	487
19.2	Data Encoding	488
19.3	Protocol Messages	512
19.4	Compression	522
19.5	Protocol and Encoding Versions	524
19.6	A Comparison with IIOP	528
Chapter 20	Ice Extension for PHP	535
20.1	Chapter Overview	535
20.2	Introduction	535
20.3	Configuration	537
20.4	Client-Side Slice-to-PHP Mapping	540
Part IV Ice Services		557
Chapter 21	IcePack	559
21.1	Chapter Overview	559
21.2	Introduction	559
21.3	Concepts	560
21.4	Ice Locator Facility	562
21.5	IcePack Registry	566
21.6	IcePack Node	570
21.7	IcePack Administration Tool	573
21.8	Deployment	576
21.9	Descriptor Reference	583
21.10	Troubleshooting	592
21.11	Summary	594
Chapter 22	Freeze	595
22.1	Chapter Overview	595
22.2	Introduction	596
22.3	The Freeze Map	596
22.4	Using a Freeze Map in the File System Server	605
22.5	The Freeze Evictor	631
22.6	Using the Freeze Evictor in a File System Server	636
22.7	Summary	655

Chapter 23	FreezeScript	657
23.1	Chapter Overview	657
23.2	Introduction	657
23.3	Database Migration	658
23.4	Transformation Descriptors	664
23.5	Using <code>transformdb</code>	677
23.6	Database Inspection	682
23.7	Using <code>dumpdb</code>	693
23.8	Descriptor Expression Language	696
23.9	Summary	700
Chapter 24	IceSSL	701
24.1	Chapter Overview	701
24.2	Introduction	701
24.3	Configuring IceSSL	703
24.4	Configuring Applications	707
24.5	Configuration Reference	709
24.6	Programming with IceSSL	718
24.7	Summary	718
Chapter 25	Glacier	719
25.1	Chapter Overview	719
25.2	Introduction	719
25.3	Using Glacier	723
25.4	Callbacks	725
25.5	The Glacier Starter	730
25.6	Glacier Security	735
25.7	Summary	740
Chapter 26	IceBox	741
26.1	Chapter Overview	741
26.2	Introduction	741
26.3	The Service Manager	742
26.4	Developing a Service	743
26.5	Starting IceBox	750
26.6	Summary	751

Chapter 27	IceStorm	753
27.1	Chapter Overview	753
27.2	Introduction	753
27.3	Concepts	755
27.4	IceStorm Interface Overview	758
27.5	Using IceStorm	760
27.6	IceStorm Administration	769
27.7	Topic Federation	770
27.8	Quality of Service	778
27.9	Configuring IceStorm	779
27.10	Summary	780
Chapter 28	IcePatch	781
28.1	Chapter Overview	781
28.2	Introduction	781
28.3	The IcePatch Server	782
28.4	The IcePatch Client	785
28.5	Developing an IcePatch Client	787
28.6	Download Totals	795
28.7	Summary	797
Appendices		799
Appendix A Slice Keywords		801

Appendix B Slice Documentation	803
B.1 Ice	803
B.2 Ice::AbortBatchRequestException	807
B.3 Ice::AdapterAlreadyActiveException	807
B.4 Ice::AdapterNotFoundException	807
B.5 Ice::AlreadyRegisteredException	807
B.6 Ice::BadMagicException	808
B.7 Ice::CloseConnectionException	808
B.8 Ice::CloseTimeoutException	809
B.9 Ice::CollocationOptimizationException	809
B.10 Ice::Communicator	809
B.11 Ice::CommunicatorDestroyedException	817
B.12 Ice::CompressionException	818
B.13 Ice::CompressionNotSupportedException	818
B.14 Ice::ConnectFailedException	818
B.15 Ice::ConnectTimeoutException	819
B.16 Ice::ConnectionLostException	819
B.17 Ice::ConnectionNotValidatedException	819
B.18 Ice::ConnectionRefusedException	820
B.19 Ice::ConnectionTimeoutException	820
B.20 Ice::Current	820
B.21 Ice::DNSException	822
B.22 Ice::DatagramLimitException	822
B.23 Ice::EncapsulationException	823
B.24 Ice::EndpointParseException	823
B.25 Ice::FacetNotExistException	823
B.26 Ice::Identity	824
B.27 Ice::IdentityParseException	825
B.28 Ice::IllegalIdentityException	825
B.29 Ice::IllegalIndirectionException	825
B.30 Ice::IllegalMessageSizeException	826
B.31 Ice::Locator	826
B.32 Ice::LocatorRegistry	827
B.33 Ice::Logger	829
B.34 Ice::MarshalException	830
B.35 Ice::MemoryLimitException	831
B.36 Ice::NegativeSizeException	831
B.37 Ice::NoEndpointException	831
B.38 Ice::NoObjectFactoryException	832
B.39 Ice::NotRegisteredException	832
B.40 Ice::ObjectAdapter	833
B.41 Ice::ObjectAdapterDeactivatedException	845

B.42	<code>Ice::ObjectAdapterIdInUseException</code>	845
B.43	<code>Ice::ObjectFactory</code>	846
B.44	<code>Ice::ObjectNotExistException</code>	847
B.45	<code>Ice::ObjectNotFoundException</code>	847
B.46	<code>Ice::OperationMode</code>	848
B.47	<code>Ice::OperationNotExistException</code>	849
B.48	<code>Ice::Plugin</code>	849
B.49	<code>Ice::PluginInitializationException</code>	850
B.50	<code>Ice::PluginManager</code>	850
B.51	<code>Ice::Process</code>	851
B.52	<code>Ice::Properties</code>	852
B.53	<code>Ice::ProtocolException</code>	856
B.54	<code>Ice::ProxyParseException</code>	857
B.55	<code>Ice::ProxyUnmarshalException</code>	857
B.56	<code>Ice::RequestFailedException</code>	858
B.57	<code>Ice::Router</code>	858
B.58	<code>Ice::ServantLocator</code>	860
B.59	<code>Ice::ServerNotFoundException</code>	862
B.60	<code>Ice::SocketException</code>	862
B.61	<code>Ice::Stats</code>	862
B.62	<code>Ice::SyscallException</code>	863
B.63	<code>Ice::TimeoutException</code>	864
B.64	<code>Ice::TwoWayOnlyException</code>	864
B.65	<code>Ice::UnknownException</code>	865
B.66	<code>Ice::UnknownLocalException</code>	865
B.67	<code>Ice::UnknownMessageException</code>	866
B.68	<code>Ice::UnknownReplyStatusException</code>	866
B.69	<code>Ice::UnknownRequestIdException</code>	866
B.70	<code>Ice::UnknownUserException</code>	867
B.71	<code>Ice::UnmarshalOutOfBoundsException</code>	867
B.72	<code>Ice::UnsupportedEncodingException</code>	867
B.73	<code>Ice::UnsupportedProtocolException</code>	868
B.74	<code>Ice::VersionMismatchException</code>	869
B.75	<code>Freeze</code>	869
B.76	<code>Freeze::Connection</code>	870
B.77	<code>Freeze::DatabaseException</code>	871
B.78	<code>Freeze::DeadlockException</code>	872
B.79	<code>Freeze::Evictor</code>	872
B.80	<code>Freeze::EvictorDeactivatedException</code>	881
B.81	<code>Freeze::EvictorIterator</code>	882
B.82	<code>Freeze::InvalidPositionException</code>	883
B.83	<code>Freeze::NoSuchElementException</code>	883

B.84	Freeze::NotFoundException	883
B.85	Freeze::ObjectRecord	884
B.86	Freeze::ServantInitializer	884
B.87	Freeze::Statistics	885
B.88	Freeze::Transaction	886
B.89	Freeze::TransactionAlreadyInProgressException	887
B.90	IceBox	887
B.91	IceBox::FailureException	887
B.92	IceBox::FreezeService	888
B.93	IceBox::Service	889
B.94	IceBox::ServiceBase	890
B.95	IceBox::ServiceManager	890
B.96	IcePack	891
B.97	IcePack::AdapterDeploymentException	891
B.98	IcePack::AdapterNotExistException	891
B.99	IcePack::Admin	892
B.100	IcePack::BadSignalException	903
B.101	IcePack::DeploymentException	903
B.102	IcePack::NodeNotExistException	904
B.103	IcePack::NodeUnreachableException	904
B.104	IcePack::ObjectDeploymentException	905
B.105	IcePack::ObjectExistsException	905
B.106	IcePack::ObjectNotExistException	905
B.107	IcePack::ParserDeploymentException	905
B.108	IcePack::Query	906
B.109	IcePack::ServerActivation	907
B.110	IcePack::ServerDeploymentException	908
B.111	IcePack::ServerDescription	908
B.112	IcePack::ServerNotExistException	910
B.113	IcePack::ServerState	910
B.114	IceSSL	912
B.115	IceSSL::CertificateException	912
B.116	IceSSL::CertificateKeyMatchException	912
B.117	IceSSL::CertificateLoadException	913
B.118	IceSSL::CertificateParseException	913
B.119	IceSSL::CertificateSignatureException	913
B.120	IceSSL::CertificateSigningException	913
B.121	IceSSL::CertificateVerificationException	914
B.122	IceSSL::CertificateVerifier	914
B.123	IceSSL::CertificateVerifierTypeException	915
B.124	IceSSL::ConfigParseException	915
B.125	IceSSL::ConfigurationLoadingException	916

B.126	<code>IceSSL::ContextException</code>	916
B.127	<code>IceSSL::ContextInitializationException</code>	916
B.128	<code>IceSSL::ContextNotConfiguredException</code>	917
B.129	<code>IceSSL::ContextType</code>	917
B.130	<code>IceSSL::Plugin</code>	918
B.131	<code>IceSSL::PrivateKeyException</code>	922
B.132	<code>IceSSL::PrivateKeyLoadException</code>	922
B.133	<code>IceSSL::PrivateKeyParseException</code>	923
B.134	<code>IceSSL::ProtocolException</code>	923
B.135	<code>IceSSL::ShutdownException</code>	923
B.136	<code>IceSSL::SslException</code>	924
B.137	<code>IceSSL::TrustedCertificateAddException</code>	924
B.138	<code>IceSSL::UnsupportedContextException</code>	925
B.139	<code>Glacier</code>	925
B.140	<code>Glacier::CannotStartRouterException</code>	925
B.141	<code>Glacier::NoSessionManagerException</code>	926
B.142	<code>Glacier::PermissionDeniedException</code>	926
B.143	<code>Glacier::PermissionsVerifier</code>	926
B.144	<code>Glacier::Router</code>	927
B.145	<code>Glacier::Session</code>	928
B.146	<code>Glacier::SessionManager</code>	928
B.147	<code>Glacier::Starter</code>	929
B.148	<code>IceStorm</code>	930
B.149	<code>IceStorm::LinkExists</code>	931
B.150	<code>IceStorm::LinkInfo</code>	931
B.151	<code>IceStorm::NoSuchLink</code>	932
B.152	<code>IceStorm::NoSuchTopic</code>	933
B.153	<code>IceStorm::Topic</code>	933
B.154	<code>IceStorm::TopicExists</code>	936
B.155	<code>IceStorm::TopicManager</code>	936
B.156	<code>IcePatch</code>	938
B.157	<code>IcePatch::BusyException</code>	938
B.158	<code>IcePatch::Directory</code>	939
B.159	<code>IcePatch::DirectoryDesc</code>	940
B.160	<code>IcePatch::File</code>	940
B.161	<code>IcePatch::FileAccessException</code>	941
B.162	<code>IcePatch::FileDesc</code>	942
B.163	<code>IcePatch::Regular</code>	942
B.164	<code>IcePatch::RegularDesc</code>	944

Appendix C Properties	945
C.1 Ice Configuration Property	945
C.2 Ice Trace Properties	946
C.3 Ice Warning Properties	948
C.4 Ice Object Adapter Properties	950
C.5 Ice Plug-in Properties	952
C.6 Ice Thread Pool Properties	953
C.7 Ice Default and Override Properties	954
C.8 Ice Miscellaneous Properties	956
C.9 IceSSL Properties	962
C.10 IceBox Properties	967
C.11 IcePack Properties	969
C.12 IceStorm Properties	977
C.13 Glacier Router Properties	979
C.14 Glacier Router Starter Properties	984
C.15 Freeze Properties	990
C.16 Properties	995
Appendix D Proxies and Endpoints	999
D.1 Proxies	999
D.2 Endpoints	1001
Bibliography	1009

Chapter 1

Introduction

1.1 Introduction

Since the mid-nineties, the computing industry has been using object-oriented middleware platforms, such as DCOM [3] and CORBA [4]. Object-oriented middleware was an important step forward toward making distributed computing available to application developers. For the first time, it was possible to build distributed applications without having to be a networking guru: the middleware platform took care of the majority of networking chores, such as *marshaling* and *unmarshaling* (encoding and decoding data for transmission), mapping logical object addresses to physical transport endpoints, changing the representation of data according to the native machine architecture of client and server, and automatically starting servers on demand.

Yet, neither DCOM nor CORBA succeeded in capturing a majority of the distributed computing market, for a number of reasons:

- DCOM was a Microsoft-only solution that could not be used in heterogeneous networks containing machines running a variety of operating systems.
- DCOM was impossible to scale to large numbers (hundreds of thousands or millions) of objects, largely due to the overhead of its distributed garbage collection mechanism.
- Although available from a variety of vendors, it was rarely possible to find a single vendor that could provide an implementation for all of the environ-

ments in a heterogeneous network. Despite much standardization effort, lack of interoperability between different CORBA implementations continued to cause problems, and source code compatibility for languages such as C or C++ was never fully achieved, usually due to vendor-specific extensions and CORBA's lack of a specification for multi-threaded environments.

- Both DCOM and CORBA suffered from excessive complexity. Becoming proficient and designing for and programming with either platform was a formidable task that took many months (or, to reach expert level, many years) to master.
- Performance issues have plagued both platforms through their respective histories. For DCOM, only one implementation was available, so shopping around for a better-performing implementation was not an option. While CORBA was available from a number of vendors, it was difficult (if not impossible) to find standards-compliant implementations that performed well, mainly due to the complexity imposed by the CORBA specification itself (which, in many cases, was feature-rich beyond need).
- In heterogeneous environments, the coexistence of DCOM and CORBA was never an easy one either: while some vendors offered interoperability products, interoperability between the two platforms was never seamless and difficult to administer, resulting in disconnected islands of different technologies.

DCOM was superseded by the Microsoft .NET platform [11] in 2002. While .NET offers more powerful distributed computing support than DCOM, it is still a Microsoft-only solution and therefore not an option for heterogeneous environments. On the other hand, CORBA has been stagnating in recent history and a number of vendors have left the market, leaving the customer with a platform that is no longer widely supported; the interest of the few remaining vendors in further standardization has waned, with the result that many defects in the CORBA specifications are not addressed, or addressed only years after they are first reported.

Simultaneously with the decline of DCOM and CORBA, a lot of interest arose in the distributed computing community around SOAP [23] and web services [24]. The idea of using the ubiquitous World Wide Web infrastructure and HTTP to develop a middleware platform was intriguing—at least in theory, SOAP and web services had the promise of becoming the lingua franca of distributed computing on the Internet. Despite much publicity and many published papers, web services have failed to deliver on that promise: as of this writing, very few commercial systems that use the web services architecture have been developed. There are a number of reasons for this:

- SOAP imposes very serious performance penalties on applications, both in terms of network bandwidth and CPU overhead, to the extent that the technology is unsuitable for many performance-critical systems.
- While SOAP provides an “on-the-wire” specification, this is insufficient for the development of realistic applications because the abstraction levels provided by the specifications are too low. While an application can cobble SOAP messages together, doing so is tedious and error-prone in the extreme.
- The lack of higher-level abstractions prompted a number of vendors to provide application development platforms that automate the development of SOAP-compliant applications. However, these development platforms, lacking any standardization beyond the protocol level, are by necessity proprietary, so applications developed with tools from one vendor cannot be used with middleware products from other vendors.
- There are serious concerns [15] about the architectural soundness of SOAP and web services. In particular, many experts have expressed concerns about the inherent lack of security of the platform.
- Web services is a technology in its infancy. Little standardization has taken place so far [24], and it appears that it will be years before standardization reaches the level of completeness that is necessary for source code compatibility and cross-vendor interoperability.

As a result, developers who are looking for a middleware platform are faced with a number of equally unpleasant options:

- Choose .NET
The most serious drawback is that non-Microsoft platforms are not supported.
- Choose CORBA
The most serious drawbacks are the high degree of complexity of an aging platform, coupled with ongoing vendor attrition.
- Choose Web Services
The most serious drawbacks are the severe inefficiencies and the need to use proprietary development platforms, as well as security issues.

These options look very much like a no-win scenario: you can either choose a platform that won’t run on anything but Microsoft architectures, or you can choose a platform that is complex and suffering from gradual abandonment, or you can choose a platform that is inefficient and, due to the lack of standardization, proprietary.

1.2 The Internet Communications Engine (Ice)

It is against this unpleasant background of choices that ZeroC, Inc. decided to develop the Internet Communications Engine, or Ice for short.¹ The main design goals of Ice are:

- Provide an object-oriented middleware platform suitable for use in heterogeneous environments.
- Provide a full set of features that support development of realistic distributed applications for a wide variety of domains.
- Avoid unnecessary complexity, making the platform easy to learn and to use.
- Provide an implementation that is efficient in network bandwidth, memory use, and CPU overhead.
- Provide an implementation that has built-in security, making it suitable for use over insecure public networks.

To be more simplistic, the Ice design goals could be stated as “Let’s build a middleware platform that is as powerful as CORBA, without making all of CORBA’s mistakes.”

1.3 Organization of this Book

This book is divided into four parts and a number of appendixes:

- Part I, Ice Overview, provides an overview of the features offered by Ice and explains the Ice object model. After reading this part, you will understand the major features and architecture of the Ice platform, its object model and request dispatch model, and know the basic steps required to build a simple application in both C++ and Java.
- Part II, Ice Core Concepts, explains the Slice definition language, and presents the C++ and Java language mappings. Part II also covers the Ice threading model and the Ice threading API. After reading this part, you will have detailed knowledge of how to specify interfaces for a distributed application, and how to implement that application in either C++ or Java.

1. The acronym “Ice” is pronounced as a single syllable, like the word for frozen water.

- Part III, Advanced Ice, presents many Ice features in detail and covers advanced aspects of server development, such as object life cycle, object location, persistence, and asynchronous method invocation and dispatch. After reading this part, you will understand the advanced features of Ice and how to effectively use them to find the correct trade-off between performance and resource consumption as appropriate for your application requirements.
- Part IV, Ice Services, covers the services provided with Ice, such as IcePack (a sophisticated deployment tool), Glacier (the Ice firewall solution), IceStorm (the Ice messaging service), and IcePatch (a software patching service).²
- The appendixes contain the Ice reference material.

1.4 Typographical Conventions

This book uses the following typographical conventions:

- Slice source code appears in **Lucida Sans Typewriter**.
- C++ or Java source code appears in **Courier**.
- File names appear in **Courier**.
- Commands appear in **Courier Bold**.

Occasionally, we present copy of an interactive session at a terminal. In such cases, we assume a Bourne shell (or one of its derivatives, such as **ksh** or **bash**). Output presented by the system is shown in **Courier**, and input is presented in **Courier Bold**, for example:

```
$ echo hello  
hello
```

Slice and C++ or Java often use the same identifiers. When we talk about an identifier in its generic, language-independent sense, we use **Lucida Sans Type-writer**. When we talk about an identifier in its language-specific (C++ or Java) sense, we use **Courier**.

2. If you notice a certain commonality in the theme of naming Ice features, it just goes to show that software developers are still inveterate punsters.

1.5 Source Code Examples

Throughout the book, we use a case study to illustrate various aspects of Ice. The case study implements a simple distributed hierarchical file system, which we progressively improve to take advantage of more sophisticated features as the book progresses. The source code for the case study in its various stages is provided with the distribution of this book. We encourage you to experiment with these code examples (as well as the many demonstration programs that ship with Ice).

1.6 Contacting the Authors

We would very much like to hear from you in case you find any bugs (however minor) in this book. We also would like to hear your opinion on the contents, and any suggestions as to how it might be improved. You can contact us via e-mail at <mailto:icebook@zeroc.com>.

1.7 Ice Support

ZeroC maintains a discussion and support forum at <http://www.zeroc.com/support.html>. You can use this forum for any questions or suggestions you may have about the Ice platform, as well as to get support for specific problems you encounter.

Part I

Ice Overview

Chapter 2

Ice Overview

2.1 Chapter Overview

In this chapter, we present a high-level overview of the Ice architecture. Section 2.2 introduces fundamental concepts and terminology, and outlines how Slice definitions, language mappings, and the Ice run time and protocol work in concert to create clients and servers. Section 2.3 briefly presents the object services provided by Ice, and Section 2.4 outlines the benefits that result from the Ice architecture. Finally, Section 2.5 presents a brief comparison of the Ice and CORBA architectures.

2.2 The Ice Architecture

2.2.1 Introduction

Ice is an object-oriented middleware platform. Fundamentally, this means that Ice provides tools, APIs, and library support for building object-oriented client–server applications. Ice applications are suitable for use in heterogeneous environments: client and server can be written in different programming languages, can run on different operating systems and machine architectures, and can communicate

using a variety of networking technologies. The source code for these applications is portable regardless of the deployment environment.

2.2.2 Terminology

Every computing technology creates its own vocabulary as it evolves. Ice is no exception. However, the amount of new jargon used by Ice is minimal. Rather than inventing new terms, we have used existing terminology as much as possible. If you have used another middleware technology, such as CORBA, in the past, you will be familiar with most of what follows. (However, we suggest you at least skim the material because a few terms used by Ice *do* differ from the corresponding CORBA terminology.)

Clients and Servers

The terms *client* and *server* are not firm designations for particular parts of an application; rather, they denote roles that are taken by parts of an application for the duration of a request:

- Clients are active entities. They issue requests for service to servers.
- Servers are passive entities. They provide services in response to client requests.

Frequently, servers are not “pure” servers, in the sense that they never issue requests and only respond to requests. Instead, servers often act as a server on behalf of some client but, in turn, act as a client to another server in order to satisfy their client’s request.

Similarly, clients often are not “pure” clients, in the sense that they only request service from an object. Instead, clients are frequently client–server hybrids. For example, a client might start a long-running operation on a server; as part of starting the operation, the client can provide a *callback object* to the server that is used by the server to notify the client when the operation is complete. In that case, the client acts as a client when it starts the operation, and as a server when it is notified that the operation is complete.

Such role reversal is common in many systems, so, frequently, client–server systems could be more accurately described as *peer-to-peer* systems.

Ice Objects

An *Ice object* is a conceptual entity, or abstraction. An Ice object can be characterized by the following points:

- An Ice object is an entity in the local or a remote address space that can respond to client requests.
- A single Ice object can be instantiated in a single server or, redundantly, in multiple servers. If an object has multiple simultaneous instantiations, it is still a single Ice object.
- Each Ice object has one or more *interfaces*. An interface is a collection of named *operations* that are supported by an object. Clients issue requests by invoking operations.
- An operation has zero or more *parameters* as well as a *return value*. Parameters and return values have a specific *type*. Parameters are named and have a direction: in-parameters are initialized by the client and passed to the server; out-parameters are initialized by the server and passed to the client. (The return value is simply a special out-parameter.)
- An Ice object has a distinguished interface, known as its *main interface*. In addition, an Ice object can provide zero or more alternate interfaces, known as *facets*. Clients can select among the facets of an object to choose the interface they want to work with.
- Each Ice object has a unique *object identity*. An object's identity is an identifying value that distinguishes the object from all other objects. The Ice object model assumes that object identities are globally unique, that is, no two objects within an Ice communication domain can have the same object identity.

In practice, you need not use object identities that are globally unique, such as UUIDs [14], only identities that do not clash within any other identity within your domain of interest. However, there are architectural advantages to using globally unique identifiers, which we explore in XREF.

Proxies

For a client to be able to contact an Ice object, the client must hold a *proxy* for the Ice object.¹ A proxy is an artifact that is local to the client's address space; it represents the (possibly remote) Ice object for the client. A proxy acts as the local

1. A proxy is the equivalent of a CORBA object reference. We use “proxy” instead of “reference” to avoid confusion: “reference” already has too many other meanings in various programming languages.

ambassador for an Ice object: when the client invokes an operation on the proxy, the Ice run time:

1. Locates the Ice object
2. Activates the Ice object's server if it is not running
3. Activates the Ice object within the server
4. Transmits any in-parameters to the Ice object
5. Waits for the operation to complete
6. Returns any out-parameters and the return value to the client (or throws an exception in case of an error)

A proxy encapsulates all the necessary information for this sequence of steps to take place. In particular, a proxy contains:

- Addressing information that allows the client-side run time to contact the correct server
- An object identity that identifies which particular object in the server is the target of a request
- An optional facet identifier that determines which particular facet of an object the proxy refers to

Stringified Proxies

The information in a proxy can be expressed as a string. For example, the string

```
SimplePrinter:default -p 10000
```

is a human-readable representation of a proxy. The Ice run time provides API calls that allow you to convert a proxy to its stringified form and vice versa. This is useful, for example, to store proxies in database tables or text files.

Provided that a client knows the identity of an Ice object and its addressing information, it can create a proxy “out of thin air” by supplying that information. In other words, no part of the information inside a proxy is considered opaque; a client needs to know only an object's identity, addressing information, and (to be able to invoke an operation) the object's type in order to contact the object.

Direct Proxies

A *direct proxy* is a proxy that embeds an object's identity, together with the *address* at which its server runs. The address is completely specified by:

- a protocol identifier (such TCP/IP or UDP)
- a protocol-specific address (such as a host name and port number)

To contact the object denoted by a direct proxy, the Ice run time uses the addressing information in the proxy to contact the server; the identity of the object is sent to the server with each request made by the client.

Indirect Proxies

An *indirect proxy* is a proxy that embeds an object's identity, together with an *object adapter name*. Note that an indirect proxy contains no addressing information. To locate the correct server, the client-side run time uses the object adapter name inside the proxy and passes it to a locator service, such as the IcePack service. In turn, the locator uses the adapter name as the key to a lookup table that contains the address of the server and returns the current server address to the client. The client-side run time now knows how to contact the server and dispatches the client request as usual.

The entire process is similar to the mapping from Internet domain names to IP address by the Domain Name Service (DNS): when we use a domain name, such as `www.zeroc.com`, to look up a web page, the host name is first resolved to an IP address behind the scenes and, once the correct IP address is known, the IP address is used to connect to the server. With Ice, the mapping is from an object adapter name to a protocol–address pair, but otherwise very similar. The client-side run time knows how to contact the IcePack service via configuration (just as web browsers know which DNS to use via configuration).

Direct Versus Indirect Binding

The process of resolving the information in a proxy to protocol–address pair is known as *binding*. Not surprisingly, *direct binding* is used for direct proxies, and *indirect binding* is used for indirect proxies.

The main advantage of indirect binding is that it allows us to move servers around (that is, change their address) without invalidating existing proxies that are held by clients. In other words, direct proxies avoid the extra lookup to locate the server but no longer work if a server is moved to different machine. On the other hand, indirect proxies continue to work even if we move (or *migrate*) a server.

Servants

As we mentioned on page 10, an Ice object is a conceptual entity that has a type, identity, and addressing information. However, client requests ultimately must end up with a concrete server-side processing entity that can provide the behavior for an operation invocation. To put this differently, a client request must ultimately

end up executing code inside the server, with that code written in a specific programming language and executing on a specific processor.

The server-side artifact that provides behavior for operation invocations is known as a *servant*. A servant provides substance for (or *incarnates*) one or more Ice objects. In practice, a servant is simply an instance of a class that is written by the server developer and that is registered with the server-side run time as the servant for one or more Ice objects. Methods on the class correspond to the operations on the Ice object's interface and provide the behavior for the operations.

A single servant can incarnate a single Ice object at a time or several Ice objects simultaneously. If the former, the identity of the Ice object incarnated by the servant is implicit in the servant. If the latter, the servant is provided the identity of the Ice object with each request, so it can decide which object to incarnate for the duration of the request.

Conversely, a single Ice object can have multiple servants. For example, we might choose to create a proxy for an Ice object with two different addresses for different machines. In that case, we will have two servers, with each server containing a servant for the same Ice object. When a client invokes an operation on such an Ice object, the client-side run time sends the request to exactly one server. In other words, multiple servants for a single Ice object allow you to build redundant systems: the client-side run time attempts to send the request to one server and, if that attempt fails, sends the request to the second server. Only if the second attempt fails is an error reported back to the client-side application code.

At-Most-Once Semantics

Ice requests have *at-most-once* semantics: the Ice run time does its best to deliver a request to the correct destination and, depending on the exact circumstances, may retry a failed request. Ice guarantees that it will either deliver the request, or, if it cannot deliver the request, inform the client with an appropriate exception; under no circumstances is a request delivered twice, that is, retries are attempted only if it is known that a previous attempt definitely failed.²

At-most-once semantics are important because they guarantee that operations that are not *idempotent* can be used safely. An idempotent operation is an operation that, if executed twice, has the same effect as if executed once. For example, `x = 1;` is an idempotent operation: if we execute the operation twice, the end

2. One exception to this rule are datagram invocations over UDP transports. For these, duplicated UDP packets can lead to a violation of at-most-once semantics.

result is the same as if we had executed it once. On the other hand, `x++ ;` is not idempotent: if we execute the operation twice, the end result is not the same as if we had executed it once.

Without at-most-once semantics, we can build distributed systems that are more robust in the presence of network failures. However, realistic systems require non-idempotent operations, so at-most-once semantics are a necessity, even though they make the system less robust in the presence of network failures. Ice permits you to mark individual operations as idempotent. For such operations, the Ice run time uses a more aggressive error recovery mechanism than for non-idempotent operations.

Synchronous Method Invocation

By default, the request dispatch model used by Ice is a synchronous remote procedure call: an operation invocation behaves like a local procedure call, that is, the client thread is suspended for the duration of the call and resumes when the call completes (and all its results are available).

Asynchronous Method Invocation

Ice also supports *asynchronous method invocation (AMI)*: clients can invoke operations *asynchronously*, that is, the client uses a proxy as usual to invoke an operation but, in addition to passing the normal parameters, also passes a *callback object* and the client invocation returns immediately. Once the operation completes, the client-side run time invokes a method on the callback object passed initially, passing the results of the operation to the callback object (or, in case of failure, passing exception information).

The server cannot distinguish an asynchronous invocation from a synchronous one—either way, the server simply sees that a client has invoked an operation on an object.

Asynchronous Method Dispatch

Asynchronous method dispatch (AMD) is the server-side equivalent of AMI. For synchronous dispatch (the default), the server-side run time up-calls into the application code in the server in response to an operation invocation. While the operation is executing (or sleeping, for example, because it is waiting for data), a thread of execution is tied up in the server; that thread is released only when the operation completes.

With asynchronous method dispatch, the server-side application code is informed of the arrival of an operation invocation. However, instead of being

forced to process the request immediately, the server-side application can choose to delay processing of the request and, in doing so, releases the execution thread for the request. The server-side application code is now free to do whatever it likes. Eventually, once the results of the operation are available, the server-side application code makes an API call to inform the server-side Ice run time that a request that was dispatched previously is now complete; at that point, the results of the operation are returned to the client.

Asynchronous method dispatch is useful if, for example, a server offers operations that block clients for an extended period of time. For example, the server may have an object with a get operation that returns data from an external, asynchronous data source and that blocks clients until the data becomes available.

With synchronous dispatch, each client waiting for data to arrive ties up an execution thread in the server. Clearly, this approach does not scale beyond a few dozen clients. With asynchronous dispatch, hundreds or thousands of clients can be blocked in the same operation invocation without tying up any threads in the server.

Another way to use asynchronous method dispatch is to complete an operation, so the results of the operation are returned to the client, but to keep the execution thread of the operation beyond the duration of the operation invocation. This allows you to continue processing after results have been returned to the client, for example, to perform cleanup or write updates to persistent storage.

Synchronous and asynchronous method dispatch are transparent to the client, that is, the client cannot tell whether a server chose to process a request synchronously or asynchronously.

Oneway Method Invocation

Clients can invoke an operation as a *oneway* operation. A oneway invocation has “best effort” semantics. For a oneway invocation, the client-side run time hands the invocation to the local transport, and the invocation completes on the client side as soon as the local transport has buffered the invocation. The actual invocation is then sent asynchronously by the operating system. The server does not reply to oneway invocations, that is, traffic flows only from client to server, but not vice versa.

Oneway invocations are unreliable. For example, the target object may not exist, in which case the invocation is simply lost. Similarly, the operation may be dispatched to a servant in the server, but the operation may fail (for example, because parameter values are invalid); if so, the client receives no notification that something has gone wrong.

Oneway invocations are possible only on operations that do not have a return value, do not have out-parameters, and do not throw user exceptions (see Chapter 4).

To the application code on the server-side, oneway invocations are transparent, that is, there is no way to distinguish a twoway invocation from a oneway invocation.

Oneway invocations are available only if the target object offers a stream-oriented transport, such as TCP/IP or SSL.

Note that, even though oneway operations are sent over a stream-oriented transport, they may be processed out of order in the server. This can happen because each invocation is dispatched in its own thread: even though the invocations are *initiated* in the order in which the invocations arrive at the server, this does not mean that they will be *processed* in that order—the vagaries of thread scheduling can result in a oneway invocation to complete before other oneway invocations that were received earlier.

Batched Oneway Method Invocation

Each oneway invocation sends a separate message to the server. For a series of short messages, the overhead of doing so is considerable: the client- and server-side run time each must switch between user mode and kernel mode for each message and, at the networking level, each message incurs the overheads of flow-control and acknowledgement.

Batched oneway invocations allow you to send a series of oneway invocations as a single message: every time you invoke a batched oneway operation, the invocation is buffered in the client-side run time. Once you have accumulated all the oneway invocations you want to send, you make a separate API call to send all the invocations at once. The client-side run time then sends all of the buffered invocations in a single message, and the server receives all of the invocations in a single message. This avoids the overhead of repeatedly trapping into the kernel for both client and server, and is much easier on the network between them because one large message can be transmitted more efficiently than many small ones.

The individual invocations in a batched oneway message are dispatched by a single thread in the order in which they were placed into the batch. This guarantees that the individual operations in a batched oneway message are processed in order in the server.

Batched oneway invocations are particularly useful for messaging services, such as IceStorm (see Chapter 27), and for fine-grained interfaces that offer set operations for small attributes.

Datagram Invocations

Datagram invocations have similar “best effort” semantics to oneway invocations. However, datagram invocations require the object to offer UDP as a transport (whereas oneway invocations require TCP/IP).

Like a oneway invocation, a datagram invocation can be made only if the operation does not have a return value, out-parameters, or user exceptions. A datagram invocation uses UDP to invoke the operation. The operation returns as soon as the local UDP stack has accepted the message; the actual operation invocation is sent asynchronously by the network stack behind the scenes.

Datagrams, like oneway invocations, are unreliable: the target object may not exist in the server, the server may not be running, or the operation may be invoked in the server but fail due to invalid parameters sent by the client. As for oneway invocations, the client receives no notification of such errors.

However, unlike oneway invocations, datagram invocations have a number of additional error scenarios:

- Individual invocations may simply be lost in the network.

This is due to the unreliable delivery of UDP packets. For example, if you invoke three operations in sequence, the middle invocation may be lost. (The same thing cannot happen for oneway invocations—because they are delivered over a connection-oriented transport, individual invocations cannot be lost.)

- Individual invocations may arrive out of order.

Again, this is due to the nature of UDP datagrams. Because each invocation is sent as a separate datagram, and individual datagrams can take different paths through network, it can happen that invocations arrive in an order that differs from the order in which they were sent.

Datagram invocations are well suited for small messages on LANs, where the likelihood of loss is small. They are also suited to situations in which low latency is more important than reliability, such as for fast, interactive internet applications.

Batched Datagram Invocations

As for batched oneway invocations, *batched datagram invocations* allow you to accumulate a number of invocations in a buffer and then send the entire buffer as a single datagram by making an API call to flush the buffer. Batched datagrams reduce the overhead of repeated system calls and allow the underlying network to operate more efficiently. However, batched datagram invocations are useful only for batched messages whose total size does not substantially exceed the PDU limit

of the network: if the size of a batched datagram gets too large, UDP fragmentation makes it more likely that one or more fragments are lost, which results in the loss of the entire batched message. However, you are guaranteed that either all invocations in a batch will be delivered, or none will be delivered. It is impossible for individual invocations within a batch to be lost.

Batched datagrams use a single thread in the server to dispatch the individual invocations in a batch. This guarantees that the invocations are made in the order in which they were queued—invocations cannot appear to be reordered in the server.

Run-Time Exceptions

Any operation invocation can raise a *run-time exception*. Run-time exceptions are pre-defined by the Ice run time and cover common error conditions, such as connection failure, connection timeout, or resource allocation failure. Run-time exceptions are presented to the application as proper C++ or Java exceptions and so integrate neatly with the native exception handling capabilities of these languages.

User Exceptions

User exceptions are used to indicate application-specific error conditions to clients. User exceptions can carry an arbitrary amount of complex data and can be arranged into inheritance hierarchies, which makes it easy for clients to handle categories of errors generically, by catching an exception that is further up the inheritance hierarchy. Like run-time exceptions, user exceptions map to native exceptions in both C++ and Java.

Properties

Much of the Ice run time is configurable via *properties*. Properties are name–value pairs, such as `Ice.Default.Protocol=tcp`. Properties are typically stored in text files and parsed by the Ice run time to configure various options, such as the tread pool size, the level of tracing, and various other configuration parameters.

2.2.3 Slice (Specification Language for Ice)

As mentioned on page 11, each Ice object has an interface with a number of operations. Interfaces, operations, and the types of data that are exchanged between client and server are defined using the *Slice language*. Slice allows you to define the client-server contract in a way that is independent of a specific programming

language, such as C++ or Java. The Slice definitions are compiled by a compiler into an API for a specific programming language, that is, the part of the API that is specific to the interfaces and types you have defined consists of generated code.

2.2.4 Language Mappings

The rules that govern how each Slice construct is translated into a specific programming language are known as *language mappings*. For example, for the C++ mapping (see Chapter 6), a Slice sequence appears as an STL vector, whereas, for the Java mapping (see Chapter 8), a Slice sequence appears as a Java array. In order to determine what the API for a specific Slice construct looks like, you only need the Slice definition and knowledge of the language mapping rules. The rules are simple and regular enough to make it unnecessary to read the generated code to work out how to use the generated API.

Of course, you are free to peruse the generated code. However, as a rule, that is inefficient because the generated code is not necessarily suitable for human consumption. We recommend that you familiarize yourself with the language mapping rules; that way, you can mostly ignore the generated code and need to refer to it only when you are interested in some specific detail.

Currently, Ice provides language mappings for C++, Java, and, for the client side, PHP (see Chapter 20).

2.2.5 Client and Server Structure

Ice clients and servers have the logical internal structure shown in Figure 2.1

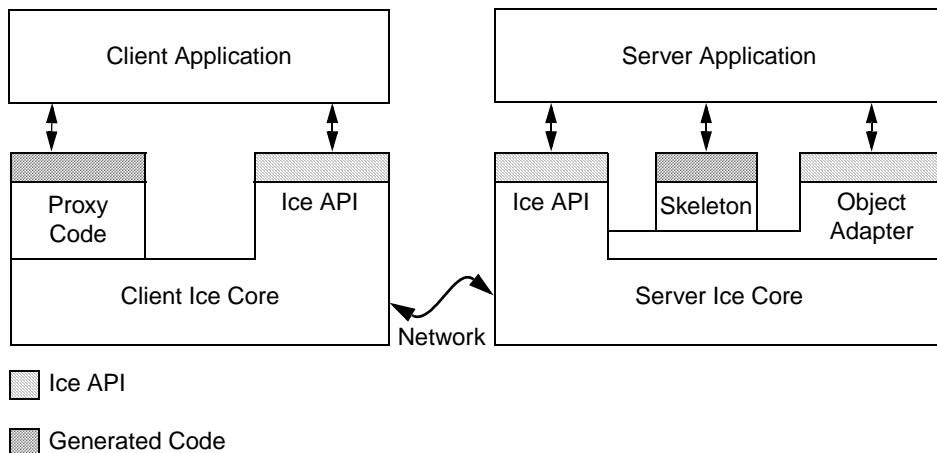


Figure 2.1. Ice Client and Server Structure

Both client and server consist of a mixture of application code, library code, and code generated from Slice definitions:

- The Ice core contains the client- and server-side run-time support for remote communication. Much of this code is concerned with the details of networking, threading, byte ordering, and many other networking-related issues that we want to keep away from application code. The Ice core is provided as a number of libraries that client and server link with.
- The generic part of the Ice core (that is, the part that is independent of the specific types you have defined in Slice) is accessed through the Ice API. You use the Ice API to take care of administrative chores, such as initializing and finalizing the Ice run time. The Ice API is identical for clients and servers (although servers use a larger part of the API than clients).
- The proxy code is generated from your Slice definitions and, therefore, specific to the types of objects and data you have defined in Slice. The proxy code has two major functions:
 - It provides a down-call interface for the client. Calling a function in the generated proxy API ultimately ends up sending an RPC message to the server that invokes a corresponding function on the target object.

- It provides *marshaling* and *unmarshaling* code.

Marshaling is the process of serializing a complex data structure, such as a sequence or a dictionary, for transmission on the wire. The marshaling code converts data into a form that is standardized for transmission and independent of the endian-ness and padding rules of the local machine.

Unmarshaling is the reverse of marshaling, that is, deserializing data that arrives over the network and reconstructing a local representation of the data in types that are appropriate for the programming language in use.

- The skeleton code is also generated from your Slice definition and, therefore, specific to the types of objects and data you have defined in Slice. The skeleton code is the server-side equivalent of the client-side proxy code: it provides an up-call interface that permits the Ice run time to transfer the thread of control to the application code you write. The skeleton also contains marshaling and unmarshaling code, so the server can receive parameters sent by the client, and return parameters and exceptions to the client.
- The object adapter is a part of the Ice API that is specific to the server side: only servers use object adapters. An object adapter has several functions:
 - The object adapter maps incoming requests from clients to specific methods on programming-language objects. In other words, the object adapter tracks which servants with what object identity are in memory.
 - The object adapter is associated with one or more transport endpoints. If more than one transport endpoint is associated with an adapter, the servants incarnating objects within the adapter can be reached via multiple transports. For example, you can associate both a TCP/IP and a UDP endpoint with an adapter, to provide alternate quality-of-service and performance characteristics.
 - The object adapter is responsible for the creation of proxies that can be passed to clients. The object adapter knows about the type, identity, and transport details of each of its objects and embeds the correct details when the server-side application code requests the creation of a proxy.

Note that, as far as the process view is concerned, there are only two processes involved: the client and the server. All the run time support for distributed communication is provided by the Ice libraries and the code that is generated from Slice definitions. (For indirect proxies, a third process, IcePack, is required to resolve proxies to transport endpoints.)

2.2.6 The Ice Protocol

Ice provides an RPC protocol that can use either TCP/IP or UDP as an underlying transport. In addition, Ice also allows you to use SSL as a transport, so all communication between client and server are encrypted.

The Ice protocol defines:

- a number of message types, such as request and reply message types,
- a protocol state machine that determines in what sequence different message types are exchanged by client and server, together with the associated connection establishment and tear-down semantics for TCP/IP,
- encoding rules that determine how each type of data is represented on the wire,
- a header for each message type that contains details such as the message type, the message size, and the protocol and encoding version in use.

Ice also supports compression on the wire: by setting a configuration parameter, you can arrange for all network traffic to be compressed to conserve bandwidth. This is useful if your application exchanges large amounts of data between client and server.

The Ice protocol is suitable for building highly-efficient event forwarding mechanisms because it permits forwarding of a message without knowledge of the details of the information inside a message. This means that messaging switches need not do any unmarshaling and remarshaling of messages—they can forward a message by simply treating it as an opaque buffer of bytes.

The Ice protocol also supports bidirectional operation: if a server wants to send a message to a callback object provided by the client, the callback can be made over the connection that was originally created by the client. This feature is especially important when the client is behind a firewall that permits outgoing connections, but not incoming connections.

2.2.7 Object Persistence

Ice has built-in object persistence service, known as *Freeze*. Freeze makes it easy to store object state in a database: you define the state stored by your objects in Slice, and the Freeze compiler generates code that stores and retrieves object state to and from a database. By default, Freeze uses Berkeley DB [18] as its database. However, if you prefer to store object state in another database, you can do so. We discuss Freeze in detail in Chapter 22.

Ice also offers a number of tools that make it easier to maintain databases and to migrate the contents of existing databases to a new schema if the type definitions of objects change. We discuss these tools in Chapter 23.

2.3 Ice Services

The Ice core provides a sophisticated client–server platform for distributed application development. However, realistic applications usually require more than just a remoting capability: typically, you also need a way to start servers on demand, distribute proxies to clients, distribute asynchronous events, configure your application, distribute patches for an application, and so on.

Ice ships with a number of services that provide these and other features. The services are implemented as Ice servers to which your application acts as a client. None of the services use Ice-internal features that are hidden from application developers so, in theory, you could develop equivalent services yourself. However, having these services available as part of the platform allows you to focus on application development instead of having to build a lot of infrastructure first. Moreover, building such services is not a trivial effort, so it pays to know what is available and use it instead of reinventing your own wheel.

2.3.1 IcePack

As mentioned on page 13, IcePack is the Ice location service that resolves a symbolic adapter name to a protocol–address pair for indirect binding. Apart from this location service, IcePack offers additional features:

- IcePack allows you to register servers for automatic start-up: instead of requiring a server to be running at the time a client issues a request, IcePack starts servers on demand, when the first client request arrives.
- IcePack supports deployment descriptors that make it easy to configure complex applications that contain several servers.
- IcePack provides a simple object lookup service that allows clients to obtain proxies for objects they are interested in.

2.3.2 IceBox

IceBox is a simple application server that can orchestrate the starting and stopping of a number of application components. Application components can be deployed as a dynamic library instead of as a process. This reduces overall system load, for example, by allowing you to run several application components in a single Java virtual machine instead of having multiple processes, each with its own virtual machine.

2.3.3 IceStorm

IceStorm is a publish–subscribe service that decouples clients and servers. Fundamentally, IceStorm acts as a distribution switch for events. Publishers send events to the service, which, in turn, passes the events to subscribers. In this way, a single event published by a publisher can be sent to multiple subscribers. Events are categorized by topic, and subscribers specify the topics they are interested in. Only events that match a subscriber’s topic are sent to that subscriber. The service permits selection of a number of quality-of-service criteria to allow applications to choose the appropriate trade-off between reliability and performance.

IceStorm is particularly useful if you have a need to distribute information to large numbers of application components. (A typical example is a stock ticker application with a large number of subscribers.) IceStorm decouples the publishers of information from subscribers and takes care of the redistribution of the published events. In addition, IceStorm can be run as a *federated* service, that is, multiple instances of the service can be run on different machines to spread the processing load over a number of CPUs.

2.3.4 IcePatch

IcePatch is a software patching service. It allows you to easily distribute software updates to clients. Clients simply connect to IcePatch and request updates for a particular application. The service automatically checks the version of the client’s software and downloads any updated application components in a compressed format to conserve bandwidth. Software patches can be secured using the Glacier service, so only authorized clients can download software updates.

2.3.5 Glacier

Glacier is the Ice firewall service: it allows clients and servers to securely communicate through a firewall without compromising security. Client-server traffic is fully encrypted using public key certificates and is bidirectional. Glacier offers support for mutual authentication as well as secure session management.

2.4 Architectural Benefits of Ice

The Ice architecture provides a number of benefits to application developers:

- Object-oriented semantics

Ice fully preserves the object-oriented paradigm “across the wire.” All operation invocations use late binding, so the implementation of an operation is chosen depending on the actual run-time (not static) type of an object.

- Support for synchronous and asynchronous messaging

Ice provides both synchronous and asynchronous operation invocation and dispatch, as well as publish–subscribe messaging via IceStorm. This allows you to choose a communication model according to the needs of your application instead of having to shoe-horn the application to fit a single model.

- Support for multiple interfaces

With facets, objects can provide multiple, unrelated interfaces while retaining a single object identity across these interfaces. This provides great flexibility, particularly as an application evolves but needs to remain compatible with older, already deployed clients.

- Machine independence

Clients and servers are shielded from idiosyncrasies of the underlying machine architecture. Issues such as byte ordering and padding are hidden from application code.

- Language independence

Client and server can be developed independently and in different programming languages (currently C++, Java, and, for the client side, PHP). The Slice definition used by both client and server establishes the interface contract between them and is the only thing they need to agree on.

- Implementation independence

Clients are unaware of how servers implement their objects. This means that the implementation of a server can be changed after clients are deployed, for example, to use a different persistence mechanism or even a different programming language.

- Operating system independence

The Ice APIs are fully portable, so the same source code compiles and runs under both Windows and UNIX.

- Threading support

The Ice run time is fully threaded and APIs are thread-safe. No effort (beyond synchronizing access to shared data) is required on part of the application developer to develop threaded, high-performance clients and servers.

- Transport independence

Ice currently offers both TCP/IP and UDP as transport protocols. Neither client nor server code are aware of the underlying transport. (The desired transport can be chosen by a configuration parameter.)

- Location and server transparency

The Ice run time takes care of locating objects and managing the underlying transport mechanism, such as opening and closing connections. Interactions between client and server appear connection-less. Via IcePack, you can arrange for servers to be started on demand if they are not running at the time a client invokes an operation. Servers can be migrated to different physical addresses without breaking proxies held by clients, and clients are completely unaware how object implementations are distributed over server processes.

- Security

Communications between client and server can be fully secured with strong encryption over SSL, so applications can use unsecured public networks to communicate securely. Via Glacier, you can implement secure forwarding of requests through a firewall, with full support for callbacks.

- Built-in persistence

With Freeze, creating persistent object implementations becomes trivial. Ice comes with built-in support for Berkeley DB [18], which is a high-performance database.

- Source code availability

The source code for Ice is available. While it is not necessary to have access to the source code to use the platform, it allows you to see how things are implemented or port the code to a new operating system.

Overall, Ice provides a state-of-the art development and deployment environment for distributed computing that is more complete than any other platform we are aware of.

2.5 A Comparison with CORBA

Obviously, Ice uses many ideas that can be found in CORBA and earlier distributed computing platforms, such as DCE [14]. In some areas, Ice is remarkably close to CORBA whereas, in others, the differences are profound and have far-reaching architectural implications. If you have used CORBA in the past, it is important to be aware of these differences.

2.5.1 Differences in the Object Model

The Ice object model, even though superficially the same, differs in a number of important points from the CORBA object model.

Type System

An Ice object, like a CORBA object, has exactly one most derived *main* interface. However, an Ice object can provide other interfaces as facets. It is important to notice that all facets of an Ice object share the same object identity, that is, the client sees a single object with multiple interfaces instead of several objects, each with a different interface.

Facets provide great architectural flexibility. In particular, they offer an approach to the versioning problem: it is easy to extend functionality in a server without breaking existing, already deployed clients by simply adding a new facet to an already existing object.

Proxy Semantics

Ice proxies (the equivalent of CORBA object references) are *not* opaque. Clients can always create a proxy without support from any other system component, as

long as they know the type and identity of the object. (For indirect binding, it is *not* necessary to be aware of the transport address of the object.)

Allowing clients to create proxies on demand has a number of advantages:

- Clients can create proxies without the need to consult an external look-up service, such as a naming service. In effect, the object identity and the object's name are considered to be one and the same. This eliminates the problems that can arise from having the contents of the naming service go out of sync with reality, and reduces the number of system components that must be functional for clients and servers to work correctly.
- Clients can easily bootstrap themselves by creating proxies to the initial objects they need. This eliminates the need for a separate bootstrap service.
- There is no need for different encodings of stringified proxies. A single, uniform representation is sufficient, and that representation is readable to humans. This avoids the complexities introduced by CORBA's three different object reference encodings (`IOR`, `corbaloc`, and `corbaname`).

Experience over many years with CORBA has shown that, pragmatically, opacity of object references is problematic: not only does it require more complex APIs and run-time support, it also gets in the way of building realistic systems. For that reason, mechanisms such as `corbaloc` and `corbaname` were added, as well as the (ill-defined) `is_equivalent` and `hash` operations for reference comparison. All of these mechanisms compromise the opacity of object references, but other parts of the CORBA platform still try to maintain the illusion of opaque references. As a result, the developer gets the worst of both worlds: references are neither fully opaque nor fully transparent—the resulting confusion and complexity are considerable.

Object Identity

The Ice object model assumes that object identities are universally unique (but without imposing this requirement on the application developer). The main advantage of universally unique object identities is that they permit you to migrate servers and to combine the objects in multiple separate servers into a single server without concerns about name collisions: if each Ice object has a unique identity, it is impossible for that identity to clash with the identity of another object in a different domain.

The Ice object model also uses *strong* object identity: it is possible to determine whether two proxies denote the same object as a local, client-side operation. (With CORBA, you must invoke operations on the remote objects to get reliable

identity comparison.) Local identity comparison is far more efficient and crucial for some application domains, such as a distributed transaction service.

2.5.2 Differences in Platform Support

CORBA, depending on which specification you choose to read, provides many of the services provided by Ice. For example, CORBA supports asynchronous method invocation and, with the component model, a form of multiple interfaces. However, the problem is that it is typically impossible to find these features in a single implementation. Too many CORBA specifications are either optional or not widely implemented so, as a developer, you are typically faced with having to choose which feature to do without.

Other features of Ice do not have direct CORBA equivalents:

- Asynchronous Method Dispatch (AMD)

The CORBA APIs do not provide any mechanism to suspend processing of an operation in the server, freeing the thread of control, and resuming processing of the operation later.

- Security

While there are many pages of specifications relating to security, most of them remain unimplemented to date. In particular, CORBA to date offers no practical solution that allows CORBA to coexist with firewalls.

- Protocol Features

The Ice protocol offers bidirectional support, which is a fundamental requirement for allowing callbacks through firewalls. (CORBA specified a bidirectional protocol at one point, but the specification was technically flawed and, to the best of our knowledge, never implemented.) In addition, Ice allows you to use UDP as well as TCP, so event distribution on reliable (local) networks can be made extremely efficient and light-weight. CORBA provides no support for UDP as a transport.

Another important feature of the Ice protocols is that all messages and data are fully encapsulated on the wire. This allows Ice to implement services such as IceStorm extremely efficiently because, to forward data, no unmarshaling and remarshaling is necessary. Encapsulation is also important for the deployment of protocol bridges, such as Glacier, because the bridge does not need to be configured with type-specific information.

2.5.3 Differences in Complexity

CORBA is known as a platform that is large and complex. This is largely a result of the way CORBA is standardized: decisions are reached by consensus and majority vote. In practice, this means that, when a new technology is being standardized, the only way to reach agreement is to accommodate the pet features of all interested parties. The result are specifications that are large, complex, and burdened with redundant or useless features. In turn, all this complexity leads to implementations that are large and inefficient. The complexity of the specifications is reflected in the complexity of the CORBA APIs: even experts with years of experience still need to work with a reference manual close at hand, and, due to this complexity, applications are frequently plagued with latent bugs that do not show up until after deployment.

CORBA's object model adds further to CORBA's complexity. For example, opaque object references force the specification of a naming service because clients must have some way to access object references. In turn, this requires the developer to learn yet another API, and to configure and deploy yet another service when, as with the Ice object model, no naming service is necessary in the first place.

One of the most infamous areas of complexity in CORBA is the C++ mapping. The CORBA C++ API is arcane in the extreme; in particular, the memory management issues of this mapping are more than what many developers are willing to endure. Yet, the code required to implement the C++ mapping is neither particularly small nor efficient, leading to binaries that are larger and require more memory at run time than they should. If you have used CORBA with C++ in the past, you will appreciate the simplicity, efficiency, and neat integration with STL of the Ice C++ mapping.

In contrast to CORBA, Ice is first and foremost a simple platform. The designers of Ice took great care to pick a feature set that is both sufficient and minimal: you can do everything you want, and you can do it with the smallest and simplest possible API. As you start to use Ice, you will appreciate this simplicity. It makes it easy to learn and understand the platform, and it leads to shorter development time with lower defect counts in deployed applications. At the same time, Ice does not compromise on features: with Ice, you can achieve everything you can achieve with CORBA and do so with less effort, less code, and less complexity. We see this as the most compelling advantage of Ice over any other middleware platform: things are simple, so simple, in fact, that you will be developing industrial-strength distributed applications after only a few days exposure to Ice.

Chapter 3

A Hello World Application

3.1 Chapter Overview

In this chapter, we will see how to create a very simple client–server application in both C++ and Java. The application enables remote printing: a client sends the text to be printed to a server, which in turn sends that text to a printer.

For simplicity (and because we do not want to concern ourselves with the idiosyncrasies of print spoolers for various platforms), our printer will simply print to a terminal instead of a real printer. This is no great loss: the purpose of the exercise is to show how a client can communicate with a server; once the thread of control has reached the server application code, that code can of course do anything it likes (including sending the text to a real printer). How to do this is independent of Ice and therefore not relevant here.

Note that much of the detail of the source code will remain unexplained for now. The intent is to show you how to get started and give you a feel for what the development environment looks like; we will provide all the detail throughout the remainder of this book.

3.2 Writing a Slice Definition

The first step in writing any Ice application is to write a Slice definition containing the interfaces that are used by the application. For our minimal printing application, we write the following Slice definition:

```
interface Printer
{
    void printString(string s);
};
```

We save this text in a file called `Printer.ice`.

Our Slice definition contains a single interface called `Printer`. For now, the interface is very simple and provides only a single operation, called `printString`. The `printString` operation accepts a string as its sole input parameter; the text of that string is what appears on the (possibly remote) printer.

3.3 Writing an Ice Application with C++

This section shows how to create an Ice application with C++. The equivalent Java version is shown in Section 3.4 on page 42.

Compiling a Slice Definition for C++

The first step in creating our C++ application is to compile our Slice definition to generate C++ proxies and skeletons. Under UNIX, you can compile the definition as follows:

```
$ slice2cpp Printer.ice
```

The `slice2cpp` compiler produces two C++ source files from this definition, `Printer.h` and `Printer.cpp`.

- `Printer.h`

The `Printer.h` header file contains C++ type definitions that correspond to the Slice definitions for our `Printer` interface. This header file must be included in both the client and the server source code.

- `Printer.cpp`

The `Printer.cpp` file contains the source code for our `Printer` interface. The generated source contains type-specific run-time support for both clients and servers. For example, it contains code that marshals parameter data (the

string passed to the `printString` operation) on the client side and unmarshals that data on the server side.

The `Printer.cpp` file must be compiled and linked into both client and server.

Writing and Compiling a Server

The source code for the server takes only a few lines and is shown in full here:

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;

class PrinterI : public Printer {
public:
    virtual void printString(const string & s,
                           const Ice::Current &);

};

void
PrinterI::
printString(const string & s, const Ice::Current &)
{
    cout << s << endl;
}

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter
            = ic->createObjectAdapterWithEndpoints(
                "SimplePrinterAdapter", "default -p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object,
                      Ice::stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();
    } catch (const Ice::Exception & e) {
        cerr << e << endl;
        status = 1;
    }
}
```

```
    } catch (const char * msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

There appears to be a lot of code here for something as simple as a server that just prints a string. Do not be concerned by this: most of the preceding code is boiler plate that never changes. For this very simple server, the code is dominated by this boiler plate.

Every Slice source file starts with an include directive for `Ice.h`, which contains the definitions for the Ice run time. We also include `Printer.h`, which was generated by the Slice compiler and contains the C++ definitions for our printer interface, and we import the contents of the `std` namespace for brevity in the code that follows:

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
```

Our server implements a single printer servant, of type `PrinterI`. Looking at the generated code in `Printer.h`, we find the following (tidied up a little to get rid of irrelevant detail):

```
class Printer : virtual public Ice::Object {
public:
    virtual void printString(const std::string &,
                           const Ice::Current & = Ice::Current()
                           ) = 0;
};
```

The `Printer` skeleton class definition is generated by the Slice compiler. (Note that the `printString` method is pure virtual so the skeleton class cannot be instantiated.) Our servant class inherits from the skeleton class to provide an implementation of the pure virtual `printString` method. (By convention, we use an `I`-suffix to indicate that the class implements an interface.)

```
class PrinterI : public Printer {
public:
    virtual void printString(const string & s,
                           const Ice::Current &);  
};
```

The implementation of the `printString` method is trivial: it simply writes its `string` argument to `stdout`:

```
void  
PrinterI::  
printString(const string & s, const Ice::Current &)  
{  
    cout << s << endl;  
}
```

Note that `printString` has a second parameter of type `Ice::Current`. As you can see from the definition of `Printer::printString`, the Slice compiler generates a default argument for this parameter, so we can leave it unused in our implementation. (We will examine the purpose of the `Ice::Current` parameter in Section 16.5.)

What follows is the server main program. Note the general structure of the code:

```
int  
main(int argc, char* argv[])  
{  
    int status = 0;  
    Ice::CommunicatorPtr ic;  
    try {  
  
        // Server implementation here...  
  
    } catch (const Ice::Exception & e) {  
        cerr << e << endl;  
        status = 1;  
    } catch (const char * msg) {  
        cerr << msg << endl;  
        status = 1;  
    }  
    if (ic)  
        ic->destroy();  
    return status;  
}
```

The body of `main` contains the declaration of two variables, `status` and `ic`. The `status` variable contains the exit status of the program and the `ic` variable, of type `Ice::CommunicatorPtr`, contains the main handle to the Ice run time.

Following these declarations is a `try` block in which we place all the server code, followed by two `catch` handlers. The first handler catches all exceptions that may be thrown by the Ice run time; the intent is that, if the code encounters an unexpected Ice run-time exception anywhere, the stack is unwound all the way back to `main`, which prints the exception and then returns failure to the operating system. The second handler catches string constants; the intent is that, if we encounter a fatal error condition somewhere in our code, we can simply throw a string literal with an error message. Again, this unwinds the stack all the way back to `main`, which prints the error message and then returns failure to the operating system.

Following the `try` block, we see a bit of cleanup code that calls the `destroy` method on the communicator (provided that the communicator was initialized). The cleanup call is outside the `try` block for a reason: we must ensure that the Ice run time is finalized whether the code terminates normally or terminates due to an exception.¹

The body of our `try` block contains the actual server code:

```
ic = Ice::initialize(argc, argv);
Ice::ObjectAdapterPtr adapter
= ic->createObjectAdapterWithEndpoints(
    "SimplePrinterAdapter", "default -p 10000");
Ice::ObjectPtr object = new PrinterI;
adapter->add(object,
    Ice::stringToIdentity("SimplePrinter"));
adapter->activate();
ic->waitForShutdown();
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice::initialize`. (We pass `argc` and `argv` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns a

1. Failure to call `destroy` on the communicator before the program exits results in undefined behavior.

smart pointer to an `Ice::Communicator` object, which is the main handle to the Ice run time.

2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are "`SimplePrinterAdapter`" (which is the name of the adapter) and "`default -p 10000`", which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string "`SimplePrinter`" is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.) The server starts to process incoming requests from clients as soon as the adapter is activated.
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice ships with such a helper class, called `Ice::Application`—see Section 10.3.1.) As far as actual application code is concerned, the server contains only a few lines: six lines for the definition of the `PrinterI` class, plus three² lines to instantiate a `PrinterI` object and register it with the object adapter.

Assuming that we have the server code in a file called `Server.cpp`, we can compile it as follows:

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Server.cpp
```

2. Well, two lines, really: printing space limitations force us to break source lines more often than you would in your actual source files.

This compiles both our application code and the code that was generated by the Slice compiler. We assume that the **ICE_HOME** environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `/opt/Ice`, set **ICE_HOME** to that path.) Depending on your platform, you may have to add additional include directives or other options to the compiler (such as an include directive for the STLport headers, or to control template instantiation); please see the demo programs that ship with Ice for the details.

Finally, we need to link the server into an executable:

```
$ c++ -o server Printer.o Server.o \
-L$ICE_HOME/lib -lIce -lIceUtil
```

Again, depending on the platform, the actual list of libraries you need to link against may be longer. The demo programs that ship with Ice contain all the detail. The important point to note here is that the Ice run time is shipped in two libraries, `libIce` and `libIceUtil`.

Writing and Compiling a Client

The client code looks very similar to the server. Here it is in full:

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;

int
main(int argc, char * argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectPrx base = ic->stringToProxy(
            "SimplePrinter:default -p 10000");
        PrinterPrx printer = PrinterPrx::checkedCast(base);
        if (!printer)
            throw "Invalid proxy";

        printer->printString("Hello World!");
    } catch (const Ice::Exception & ex) {
        cerr << ex << endl;
        status = 1;
    } catch (const char * msg) {
        cerr << msg << endl;
```

```
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

Note that the overall code layout is the same as for the server: we include the headers for the Ice run time and the header generated by the Slice compiler, and we use the same `try` block and `catch` handlers to deal with errors.

The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice::initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 21.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrx::checkedCast`. A checked cast sends a message to the server, effectively asking “is this a proxy for a `Printer` interface?” If so, the call returns a proxy to a `Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns a null proxy.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored “Hello World!” string. The server prints that string on its terminal.

Compiling and linking the client looks much the same as for the server:

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Client.cpp
$ c++ -o client Printer.o Client.o -L$ICE_HOME/lib -lIce -lIceUtil
```

Running Client and Server

To run client and server, we first start the server in a separate window:

```
$ ./server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ ./client  
$
```

The client runs and exits without producing any output; however, in the server window, we see the "Hello World!" that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Section 10.3.1.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get:

```
Network.cpp:471: Ice::ConnectFailedException:  
connect failed: Connection refused
```

Note that, to successfully run client and server, you will have to set some platform-dependent environment variables. For example, under Linux, you need to add the Ice library directory to your **LD_LIBRARY_PATH**. Please have a look at the demo applications that ship with Ice for the details for your platform.

3.4 Writing an Ice Application with Java

This section shows how to create an Ice application with Java. The equivalent C++ version is shown in Section 3.3 on page 34.

Compiling a Slice Definition for Java

The first step in creating our Java application is to compile our Slice definition to generate Java proxies and skeletons. Under UNIX, you can compile the definition as follows:³

```
$ mkdir generated  
$ slice2java --output-dir generated Printer.ice
```

3. Whenever we show UNIX commands in this book, we assume a Bourne or Bash shell. The commands for Windows are essentially identical and therefore not shown.

The **--output-dir** option instructs the compiler to place the generated files into the generated directory. This avoids cluttering the working directory with the generated files. The **slice2java** compiler produces a number of Java source files from this definition. The exact contents of these files do not concern us for now—they contain the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing and Compiling a Server

To implement our `Printer` interface, we must create a servant class. By convention, servant classes use the name of their interface with an `I`-suffix, so our servant class is called `PrinterI` and placed into a source file `PrinterI.java`:

```
public class PrinterI extends _PrinterDisp {
    public void
    printString(String s, Ice.Current current)
    {
        System.out.println(s);
    }
}
```

The `PrinterI` class inherits from a base class called `_PrinterDisp`, which is generated by the **slice2java** compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter. We will see its purpose in detail in Section 16.5.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code is in a source file called `Server.java`, shown in full here:

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            Ice.ObjectAdapter adapter
                = ic.createObjectAdapterWithEndpoints(
                    "SimplePrinterAdapter", "default -p 10000");
            Ice.Object object = new PrinterI();
            adapter.add(
```

```
        object,
        Ice.Util.stringToIdentity("SimplePrinter"));
    adapter.activate();
    ic.waitForShutdown();
} catch (Ice.LocalException e) {
    e.printStackTrace();
    status = 1;
} catch (Exception e) {
    System.err.println(e.getMessage());
    status = 1;
} finally {
    if (ic != null)
        ic.destroy();
}
System.exit(status);
}
}
```

Note the general structure of the code:

```
public class Server {
    public static void
    main(String[] args) {
        int status = 0;
        Ice.Communicator ic = null;
        try {

            // Server implementation here...

        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        } finally {
            if (ic != null)
                ic.destroy();
        }
        System.exit(status);
    }
}
```

The body of `main` contains a `try` block in which we place all the server code, followed by two `catch` blocks. The first block catches all exceptions that may be thrown by the Ice run time; the intent is that, if the code encounters an unexpected

Ice run-time exception anywhere, the stack is unwound all the way back to `main`, which prints the exception and then returns failure to the operating system. The second block catches `Exception` exceptions; the intent is that, if we encounter a fatal error condition somewhere in our code, we can simply throw an exception with an error message. Again, this unwinds the stack all the way back to `main`, which prints the error message and then returns failure to the operating system.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results. Placing the call to `destroy` into a `finally` block guarantees that the communicator will be correctly destroyed, regardless of any exceptions in the preceding `try` block.

The body of our `try` block contains the actual server code:

```
ic = Ice.Util.initialize(args);
Ice.ObjectAdapter adapter
    = ic.createObjectAdapterWithEndpoints(
        "SimplePrinterAdapter", "default -p 10000");
Ice.Object object = new PrinterI();
adapter.add(
    object,
    Ice.Util.stringToIdentity("SimplePrinter"));
adapter.activate();
ic.waitForShutdown();
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice::Communicator` reference, which is the main handle to the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated,

plus an identifier. In this case, the string "SimplePrinter" is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)

5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice ships with such a helper class, called `Ice.Application`—see Section 12.3.1.) As far as actual application code is concerned, the server contains only a few lines: seven lines for the definition of the `PrinterI` class, plus four⁴ lines to instantiate a `PrinterI` object and register it with the object adapter.

We can compile the server code as follows:

```
$ mkdir classes
$ javac -d classes -classpath classes:$ICEJ_HOME/lib/Ice.jar \
-source 1.4 Server.java PrinterI.java generated/*.java
```

This compiles both our application code and the code that was generated by the Slice compiler. We assume that the `ICEJ_HOME` environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `/opt/Ice/j`, set `ICEJ_HOME` to that path.) Note that Ice for Java uses the `ant` build environment to control building of source code. (`ant` is similar to `make`, but more flexible for Java applications.) You can have a look at the demo code that ships with Ice to see how to use this tool.

Writing and Compiling a Client

The client code, in `Client.java`, looks very similar to the server. Here it is in full:

4. Well, two lines, really: printing space limitations force us to break source lines more often than you would in your actual source files.

```
public class Client {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            Ice.ObjectPrx base = ic.stringToProxy(
                "SimplePrinter:default -p 10000");
            PrinterPrx printer
                = PrinterPrxHelper.checkedCast(base);
            if (printer == null)
                throw new Error("Invalid proxy");

            printer.printString("Hello World!");
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        } finally {
            if (ic != null)
                ic.destroy();
        }
        System.exit(status);
    }
}
```

Note that the overall code layout is the same as for the server: we use the same `try`, `catch`, and `finally` blocks to deal with errors. The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.Util.initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 21.)

3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrxHelper.checkedCast`. A checked cast sends a message to the server, effectively asking “is this a proxy for a `Printer` interface?” If so, the call returns a proxy of type `Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored “Hello World!” string. The server prints that string on its terminal.

Compiling the client looks much the same as for the server:

```
$ javac -d classes -classpath classes:$ICEJ_HOME/lib/Ice.jar \
-source 1.4 Client.java PrinterI.java generated/*.java
```

Running Client and Server

To run client and server, we first start the server in a separate window:

```
$ java Server
```

At this point, we won’t see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ java Client
$
```

The client runs and exits without producing any output; however, in the server window, we see the “Hello World!” that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Chapter 12.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice.ConnectFailedException
    at IceInternal.Network.doConnect(Network.java:201)
    at IceInternal.TcpConnector.connect(TcpConnector.java:26)
    at
IceInternal.OutgoingConnectionFactory.create(OutgoingConnectionFactory.java:80)
```

```
        at Ice._ObjectDelM.setup(_ObjectDelM.java:251)
        at Ice.ObjectPrxHelper.__getDelegate(ObjectPrxHelper.java:
642)
        at Ice.ObjectPrxHelper.ice_isA(ObjectPrxHelper.java:41)
        at Ice.ObjectPrxHelper.ice_isA(ObjectPrxHelper.java:30)
        at PrinterPrxHelper.checkedCast(Unknown Source)
        at Client.main(Unknown Source)
Caused by: java.net.ConnectException: Connection refused
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
        )
        at
sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:
518)
        at IceInternal.Network.doConnect(Network.java:173)
        ... 8 more
```

Note that, to successfully run client and server, your **CLASSPATH** must include the Ice library and the classes directory, for example:

```
$ export CLASSPATH=$CLASSPATH:./classes:$ICEJ_HOME/lib/Ice.jar
```

Please have a look at the demo applications that ship with Ice for the details for your platform.

3.5 Summary

This chapter presented a very simple (but complete) client and server. As we saw, writing an Ice application involves the following steps:

1. Write a Slice definition and compile it.
2. Write a server and compile it.
3. Write a client and compile it.

If someone else has written the server already and you are only writing a client, you do not need to write the Slice definition, only compile it (and, obviously, you do not need to write the server in that case).

Do not be concerned if, at this point, much appears unclear. The intent of this chapter is to give you an idea of the development process, not to explain the Ice APIs in intricate detail. We will cover all the detail throughout the remainder of this book.

Part II

Ice Core Concepts

Chapter 4

The Slice Language

4.1 Chapter Overview

In this chapter we present the Slice language. We start by discussing the role and purpose of Slice, explaining how language-independent specifications are compiled for particular implementation languages to create actual implementations. Sections 4.8 and 4.9 cover the core Slice concepts of interfaces, operations, exceptions, and inheritance. These concepts have profound influence on the behavior of a distributed system and should be read in detail.

4.2 Introduction

Slice¹ (Specification Language for Ice) is the fundamental abstraction mechanism for separating object interfaces from their implementations. Slice establishes a contract between client and server that describes the types and object interfaces used by an application. This description is independent of the implementation language, so it does not matter whether the client is written in the same language as the server.

1. Even though Slice is an acronym, it is pronounced as single syllable, like a slice of bread.

Slice definitions are compiled for a particular implementation language by a compiler. The compiler translates the language-independent definitions into language-specific type definitions and APIs. These types and APIs are used by the developer to provide application functionality and to interact with Ice. The translation algorithms for various implementation languages are known as *language mappings*. Currently, Ice defines language mappings for C++ and Java.

Because Slice describes interfaces and types (but not implementations), it is a purely declarative language; there is no way to write executable statements in Slice.

Slice definitions focus on object interfaces, the operations supported by those interfaces, and exceptions that may be raised by operations. In addition, Slice offers features for object persistence (see Chapter 22). This requires quite a bit of supporting machinery; in particular, quite a bit of Slice is concerned with the definition of data types. This is because data can be exchanged between client and server only if their types are defined in Slice. You cannot exchange arbitrary C++ data between client and server because it would destroy the language independence of Ice. However, you can always create a Slice type definition that corresponds to the C++ data you want to send, and then you can transmit the Slice type.

We present the full syntax and semantics of Slice here. Because much of Slice is based on C++ and Java, we focus on those areas where Slice differs from C++ or Java or constrains the equivalent C++ or Java feature in some way. Slice features that are identical to C++ and Java are mentioned mostly by example.

4.3 Compilation

A Slice compiler produces source files that must be combined with application code to produce client and server executables.

The outcome of the development process is a client executable and a server executable. These executables can be deployed anywhere, whether the target environments use the same or different operating systems and whether the executables are implemented using the same or different languages. The only constraint is that the host machines must provide the necessary run-time environment, such as any required dynamic libraries, and that connectivity can be established between them.

4.3.1 Single Development Environment for Client and Server

Figure 4.1 shows the situation when both client and server are developed in C++. The Slice compiler generates two files from a Slice definition in a source file `Printer.ice`: a header file (`Printer.h`) and a source file (`Printer.cpp`).

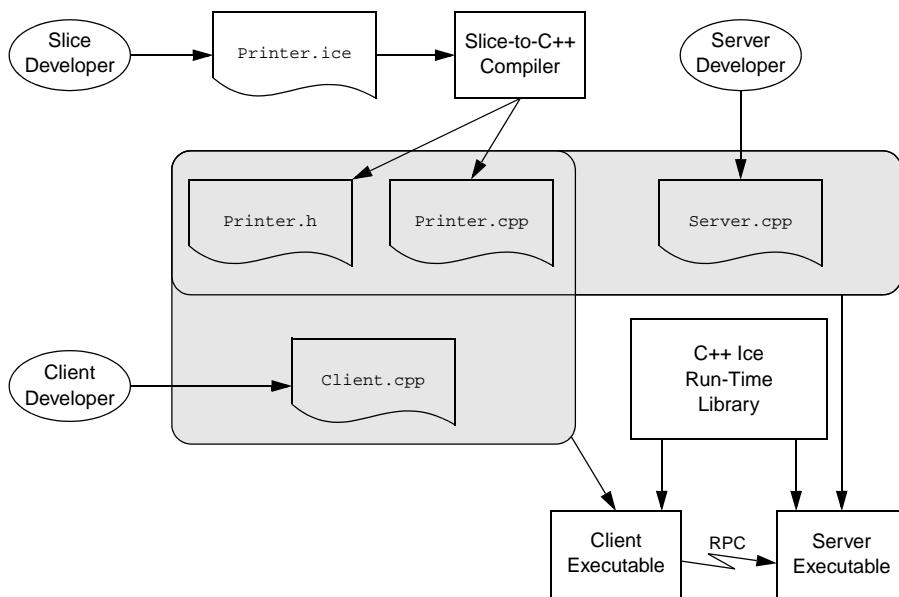


Figure 4.1. Development process if client and server share the same development environment.

- The `Printer.h` header file contains definitions that correspond to the types used in the Slice definition. It is included in the source code of both client and server to ensure that client and server agree about the types and interfaces used by the application.
- The `Printer.cpp` source file provides an API to the client for sending messages to remote objects. The client source code (`Client.cpp`, written by the client developer) contains the client-side application logic. The generated source code and the client code are compiled and linked into the client executable.

The `Printer.cpp` source file also contains source code that provides an up-call interface from the Ice run time into the server code written by the developer and provides the connection between the networking layer of Ice and the

application code. The server implementation file (`Server.cpp`, written by the server developer) contains the server-side application logic (the object implementations, properly termed *servants*). The generated source code and the implementation source code are compiled and linked into the server executable.

Both client and server also link with an Ice library that provides the necessary run-time support.

You are not limited to a single implementation of a client or server. For example, you can build multiple servers, each of which implements the same interfaces but uses different implementations (for example, with different performance characteristics). Multiple such server implementations can coexist in the same system. This arrangement provides one fundamental scalability mechanism in Ice: if you find that a server process starts to bog down as the number of objects increases, you can run an additional server for the same interfaces on a different machine. Such *federated* servers provide a single logical service that is distributed over a number of processes on different machines. Each server in the federation implements the same interfaces but hosts different object instances. (Of course, federated servers must somehow ensure consistency of any databases they share across the federation.)

Ice also provides support for *replicated* servers. Replication permits multiple servers to each implement the same set of object instances. This improves performance and scalability (because client load can be shared over a number of servers) as well as redundancy (because each object is implemented in more than one server).

4.3.2 Different Development Environments for Client and Server

Client and server cannot share any source or binary components if they are developed in different languages. For example, a client written in Java cannot include a C++ header file.

Figure 4.2 shows the situation when a client written in Java and the corresponding server is written in C++. In this case, the client and server developers are completely independent, and each uses his or her own development environment

and language mapping. The only link between client and server developers is the Slice definition each one uses.

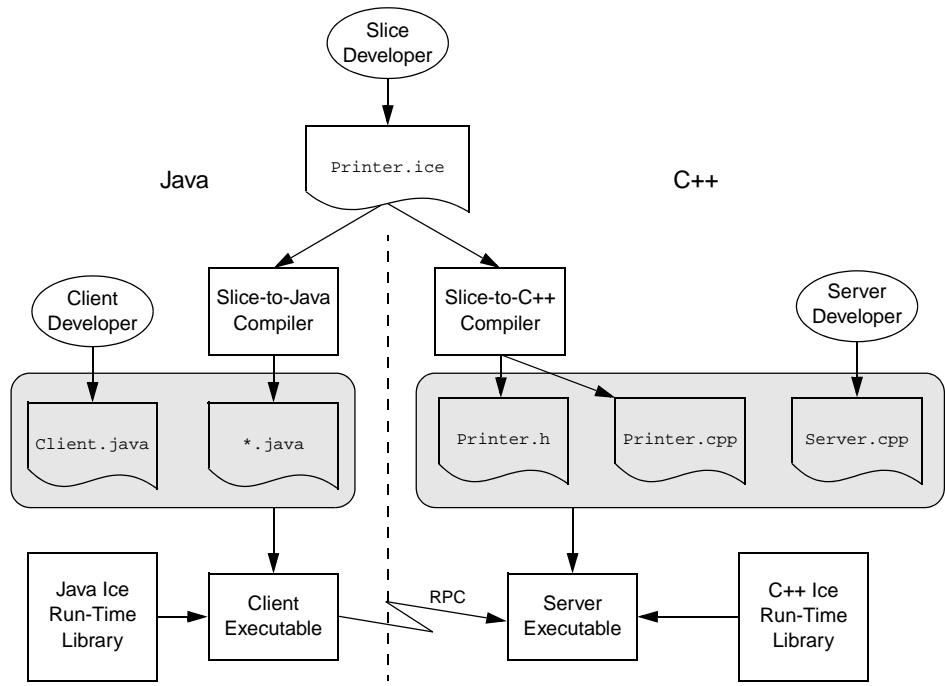


Figure 4.2. Development process for different development environments.

For Java, the slice compiler creates a number of files whose names depend on the names of various Slice constructs. (These files are collectively referred to as `*.java` in Figure 4.2.)

4.4 Source Files

Slice defines a number of rules for the naming and contents of Slice source files.

4.4.1 File Naming

Files containing Slice definitions must end in a `.ice` file extension, for example, `Clock.ice` is a valid file name. Other file extensions are rejected by the compilers.

For case-insensitive file systems (such as DOS), the file extension may be written as uppercase or lowercase, so `Clock.ICE` is legal. For case-sensitive file systems (such as UNIX), `Clock.ICE` is illegal. (The extension must be in lowercase.)

4.4.2 File Format

Slice is a free-form language so you can use spaces, horizontal and vertical tab stops, form feeds, and newline characters to lay out your code in any way you wish. (White space characters are token separators). Slice does not attach semantics to the layout of a definition. You may wish to follow the style we have used for the Slice examples throughout this book.

4.4.3 Preprocessing

Slice supports the `#ifndef`, `#define`, `#endif`, and `#include` preprocessing directives. Their use is severely restricted:

- The only purpose for which `#ifndef`, `#define`, and `#endif` directives can be used is to create a double-include block. For example:

```
#ifndef _CLOCK_ICE
#define _CLOCK_ICE

// #include directives here...

// Definitions here...

#endif _CLOCK_ICE
```

- `#include` directives can appear only at the beginning of a Slice source file, that is, they must precede all other Slice definitions. In addition, only `#include` directives using the `<>`-syntax to specify a file name are permitted; use of the `""`-syntax is not supported. For example:

```
#include <File1.ice>    // OK
#include "File2.ice"     // Not supported!
```

Use of these preprocessing directives for any other purpose, or use of other C++ preprocessing directives is not supported.² (This includes preprocessor features such as line continuation with a \ character, token pasting, and macro expansion.)

#include directives permit a Slice definition to use types defined in a different source file. The Slice compilers parse all of the code in a source file, including the code in #included files. However, the compilers generate code only for the top-level file(s) nominated on the command line. You must separately compile #included files to obtain generated code for all the files that make up your Slice definition.

We strongly recommend that you use a double-include guard in all your Slice definitions to prevent inclusion of a file more than once, as shown on page 58.

4.4.4 Definition Order

Slice constructs, such as modules, interfaces, or type definitions, can appear in any order you prefer. However, identifiers must be declared before they can be used.

4.5 Lexical Rules

Slice's lexical rules are very similar to those of C++ and Java, except for some differences for identifiers.

4.5.1 Comments

Slice definitions permit both the C and the C++ style of writing comments:

```
/*
 * C-style comment.
 */
// C++-style comment extending to the end of this line.
```

2. At the time of writing, Slice is actually preprocessed by the C++ preprocessor. However, use of the preprocessor may be replaced by a different mechanism in a future release; if so, only the described preprocessing directives and uses will actually be supported.

4.5.2 Keywords

Slice uses a number of keywords, which must be spelled in lowercase. For example, `class` and `dictionary` are keywords and must be spelled as shown. There are two exceptions to this lowercase rule: `Object` and `LocalObject` are keywords and must be capitalized as shown. You can find a full list of Slice keywords in Appendix A.

4.5.3 Identifiers

Identifiers begin with an alphabetic character followed by any number of alphabetic characters or digits. Slice identifiers are restricted to the ASCII range of alphabetic characters and cannot contain non-English letters, such as Å.

(Supporting non-ASCII identifiers would make it very difficult to map Slice to target languages that lack support for this feature.)

Unlike C++ identifiers, Slice identifiers cannot contain underscores. This restriction may seem draconian at first but is necessary: by reserving underscores, the various language mappings gain a namespace that cannot clash with legitimate Slice identifiers. That namespace can then be used to hold language-native identifiers that are derived from Slice identifiers without fear of clashing with another, legitimate Slice identifier that happens to be the same as one of the derived identifiers.

Case Sensitivity

Identifiers are case-insensitive but must be capitalized consistently. For example, `TimeOfDay` and `TIMEOFDAY` are considered the same identifier within a scope. However, Slice enforces consistent capitalization. After you have introduced an identifier, you must capitalize it consistently throughout; otherwise, the compiler will reject it as illegal. This rule exists to permit mappings of Slice to languages that ignore case in identifiers as well as to languages that treat differently capitalized identifiers as distinct.

Identifiers That Are Keywords

You can define Slice identifiers that are keywords in one or more implementation languages. For example, `switch` is a perfectly good Slice identifier but is a C++ and Java keyword. Language mappings define rules for dealing with such identifiers. The solution typically involves using a prefix to map away from the keyword. For example, the Slice identifier `switch` is mapped to `_cpp_switch` in C++ and `_switch` in Java.

The rules for dealing with keywords can result in hard-to-read source code. Identifiers such as `native`, `throw`, or `export` will clash with C++ or Java keywords (or both). To make life easier for yourself and others, try to avoid Slice identifiers that are implementation language keywords. Keep in mind that mappings for languages other than C++ and Java may be added in the future. While it is not reasonable to expect you to compile a list of all keywords in all popular programming languages, you should make an attempt to avoid at least common keywords. Slice identifiers such as `self`, `import`, and `while` are definitely not a good idea.

Escaped Identifiers

It is possible to use a Slice keyword as an identifier by prefixing the keyword with a backslash, for example:

```
struct dictionary {      // Error!
    ...
};

struct \dictionary {     // OK
    ...
};
```

The backslash escapes the usual meaning of a keyword; in the previous example, `\dictionary` is treated as the identifier `dictionary`. The escape mechanism exists to permit keywords to be added to the Slice language over time with minimal disruption to existing specifications: if a pre-existing specification happens to use a newly-introduced keyword, that specification can be fixed by simply prepending a backslash to the new keyword. Note that, as a matter of style, you should avoid using Slice keywords as identifiers (even though the backslash escapes allow you to do this).

Reserved Identifiers

Slice reserves the identifier `Ice` and all identifiers beginning with `Ice` (in any capitalization) for the Ice implementation. For example, if you try to define a type named `Icecream`, the Slice compiler will issue an error message.³

3. You can suppress this behavior by using the `--ice` compiler option, which enables definition of identifiers beginning with `Ice`. However, do not use this option unless you are compiling the Slice definitions for the Ice run time itself.

Slice identifiers ending in any of the suffixes `Helper`, `Holder`, `Prx`, and `Ptr` are also reserved. These endings are used by the Java and C++ language mappings and are reserved to prevent name clashes in the generated code.

4.6 Basic Slice Types

Slice provides a number of built-in basic types, shown in Table 4.1.

Table 4.1. Slice basic types.

Type	Range	Size
<code>bool</code>	<code>false</code> or <code>true</code>	Unspecified
<code>byte</code>	$-128\text{--}127^{\text{a}}$	≥ 8 bits
<code>short</code>	$-2^{15}\text{ to }2^{15}\text{--}1$	≥ 16 bits
<code>int</code>	$-2^{31}\text{ to }2^{31}\text{--}1$	≥ 32 bits
<code>long</code>	$-2^{63}\text{ to }2^{63}\text{--}1$	≥ 64 bits
<code>float</code>	IEEE single-precision	≥ 32 bits
<code>double</code>	IEEE double-precision	≥ 64 bits
<code>string</code>	All Unicode characters, excluding the character with all bits zero.	Variable-length

a. Or 0–255, depending on the language mapping

All the basic types (except `byte`) are subject to changes in representation as they are transmitted between clients and servers. For example, a `long` value is byte-swapped when sent from a little-endian to a big-endian machine. Similarly, strings undergo translation in representation if they are sent, for example, from an EBCDIC to an ASCII implementation, and the characters of a string may also change in size. (Not all architectures use 8-bit characters). However, these changes are transparent to the programmer and do exactly what is required.

4.6.1 Integer Types

Slice provides integer types `short`, `int`, and `long`, with 16-bit, 32-bit, and 64-bit ranges, respectively. Note that, on some architectures, any of these types may be mapped to a native type that is wider. Also note that no unsigned types are provided. (This choice was made because unsigned types are difficult to map into languages without native unsigned types, such as Java. In addition, the unsigned integers add little value to a language. See [9] for a good treatment of the topic.)

4.6.2 Floating-Point Types

These types follow the IEEE specification for single- and double-precision floating-point representation [6]. If an implementation cannot support IEEE format floating-point values, the Ice run time converts values into the native floating-point representation (possibly at a loss of precision or even magnitude, depending on the capabilities of the native floating-point format).

4.6.3 Strings

Slice strings use the Unicode character set. The only character that cannot appear inside a string is the zero character.⁴

The Slice data model does *not* have the concept of a null string (in the sense of a C++ null pointer). This decision was made because null strings are difficult to map to languages without direct support for this concept (such as Python). Do not design interfaces that depend on a null string to indicate “not there” semantics. If you need the notion of an optional string, use a class (see Section 4.9), a sequence of strings (see Section 4.7.3), or use an empty string to represent the idea of a null string. (Of course, the latter assumes that the empty string is not otherwise used as a legitimate string value by your application.)

4.6.4 Booleans

Boolean values can have only the values `false` and `true`. Language mappings use the corresponding native boolean type if one is available.

4. This decision was made as a concession to C++, with which it becomes impossibly difficult to manipulate strings with embedded zero characters using standard library routines, such as `strlen` or `strcat`.

4.6.5 Bytes

The Slice type `byte` is an (at least) 8-bit type that is guaranteed not to undergo any changes in representation as it is transmitted between address spaces. This guarantee permits exchange of binary data such that it is not tampered with in transit. All other Slice types are subject to changes in representation during transmission.

4.7 User-Defined Types

In addition to providing the built-in basic types, Slice allows you to define complex types: enumerations, structures, sequences, and dictionaries.

4.7.1 Enumerations

A Slice enumerated type definition looks like the C++ version:

```
enum Fruit { Apple, Pear, Orange };
```

This definition introduces a type named `Fruit` that becomes a new type in its own right. Slice does not define how ordinal values are assigned to enumerators. For example, you cannot assume that the enumerator `Orange` will have the value 2 in different implementation languages. Slice guarantees only that the ordinal values of enumerators increase from left to right, so `Apple` compares less than `Pear` in all implementation languages.

Unlike C++, Slice does not permit you to control the ordinal values of enumerators (because many implementation languages do not support such a feature):

```
enum Fruit { Apple = 0, Pear = 7, Orange = 2 }; // Syntax error
```

In practice, you do not care about the values used for enumerators as long as you do not transmit the *ordinal value* of an enumerator between address spaces. For example, sending the value 0 to a server to mean `Apple` can cause problems because the server may not use 0 to represent `Apple`. Instead, simply send the value `Apple` itself. If `Apple` is represented by a different ordinal value in the receiving address space, that value will be appropriately translated by the Ice run time.

As with C++, Slice enumerators enter the enclosing namespace, so the following is illegal:

```
enum Fruit { Apple, Pear, Orange };
enum ComputerBrands { Apple, IBM, Sun, HP }; // Apple redefined
```

Slice does not permit empty enumerations.

4.7.2 Structures

Slice supports structures containing one or more named members of arbitrary type, including user-defined complex types. For example:

```
struct TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};
```

As in C++, this definition introduces a new type called `TimeOfDay`. Structure definitions form a namespace, so the names of the structure members need to be unique only within their enclosing structure.

Data member definitions using a named type are the only construct that can appear inside a structure. It is impossible to, for example, define a structure inside a structure:

```
struct TwoPoints {
    struct Point {        // Illegal!
        short x;
        short y;
    };
    Point     coord1;
    Point     coord2;
};
```

This rule applies to Slice in general: type definitions cannot be nested (except for modules, which do support nesting—see Section 4.11). The reason for this rule is that nested type definitions can be difficult to implement for some target languages and, even if implementable, greatly complicate the scope resolution rules. For a specification language, such as Slice, nested type definitions are unnecessary—you can always write the above definitions as follows (which is stylistically cleaner as well):

```
struct Point {
    short x;
    short y;
};
```

```
struct TwoPoints {      // Legal (and cleaner!)
    Point coord1;
    Point coord2;
};
```

4.7.3 Sequences

Sequences are variable-length vectors of elements:

```
sequence<Fruit> FruitPlatter;
```

A sequence can be empty—that is, it can contain no elements, or it can hold any number of elements up to the memory limits of your platform.

Sequences can contain elements that are themselves sequences. This arrangement allows you to create lists of lists:

```
sequence<FruitPlatter> FruitBanquet;
```

Sequences are used to model a variety of collections, such as vectors, lists, queues, sets, bags, or trees. (It is up to the application to decide whether or not order is important; by discarding order, a sequence serves as a set or bag.)

One particular use of sequences has become idiomatic, namely, the use of a sequence to indicate an optional value. For example, we might have a `Part` structure that records the details of the parts that go into a car. The structure could record things such as the name of the part, a description, weight, price, and other details. Spare parts commonly have a serial number, which we can model as a `long` value. However, some parts, such as simple screws, often do not have a serial number, so what are we supposed to put into the serial number field of a screw? There a number of options for dealing with this situation:

- Use a sentinel value, such as zero, to indicate the “no serial” number condition.

This approach is workable, provided that a sentinel value is actually available. While it may seem unlikely that anyone would use a serial number of zero for their parts, it is not impossible. And, for other values, such as a temperature value, all values in the range of their type can be legal, so no sentinel value is available.

- Change the type of the serial number from `long` to `string`.

Strings come with their own built-in sentinel value, namely, the empty string so we can use an empty string to indicate the “no serial number” case. This is

workable, but leaves a bad taste in most people's mouth: we should not have to change the natural data type of something to `string` just so we get a sentinel value.

- Add an indicator as to whether the contents of the serial number are valid:

```
struct Part {  
    string name;  
    string description;  
    // ...  
    bool serialIsValid; // true if part has serial number  
    long serialNumber;  
};
```

This is distasteful to most people and guaranteed to get you into trouble eventually: sooner or later, some programmer will forget to check whether the serial number is valid before using it and create havoc.

- Use a sequence to model the optional field.

This technique uses the following convention:

```
sequence<long> SerialOpt;  
  
struct Part {  
    string name;  
    string description;  
    // ...  
    SerialOpt serialNumber; // optional: zero or one element  
};
```

By convention, the `Opt` suffix is used to indicate that the sequence is used to model an optional value. If the sequence is empty, the value is obviously not there; if it contains a single element, that element is the value. The obvious drawback of this scheme is that someone could put more than one element into the sequence. This could be rectified by adding a special-purpose Slice construct for optional values. However, optional values are not used frequently enough to justify the complexity of adding a dedicated language feature. (As we will see in Section 4.9, you can also use class hierarchies to model optional fields.)

4.7.4 Dictionaries

A dictionary is a mapping from a key type to a value type. For example:

```

struct Employee {
    long number;
    string firstName;
    string lastName;
};

dictionary<long, Employee> EmployeeMap;

```

This definition creates a dictionary named `EmployeeMap` that maps from an employee number to a structure containing the details for an employee. Whether or not the key type (the employee number, of type `long` in this example) is also part of the value type (the `Employee` structure in this example) is up to you—as far as Slice is concerned, there is no need to include the key as part of the value.

Dictionaries can be used to implement sparse arrays, or any lookup data structure with non-integral key type. Even though a sequence of structures containing key–value pairs could be used to model the same thing, a dictionary is more appropriate:

- A dictionary clearly signals the intent of the designer, namely, to provide a mapping from a domain of values to a range of values. (A sequence of structures of key–value pairs does not signal that same intent as clearly.)
- At the programming language level, sequences are implemented as vectors (or possibly lists), that is, they are not well suited to model sparsely populated domains and require a linear search to locate an element with a particular value. On the other hand, dictionaries are implemented as a data structure (typically a hash table or red-black tree) that supports efficient searching in $O(\log n)$ average time or better.

The key type of a dictionary need not be an integral type. For example, we could use the following definition to translate the names of the days of the week:

```
dictionary<string, string> WeekdaysEnglishToGerman;
```

The server implementation would take care of initializing this map with the key–value pairs `Monday–Montag`, `Tuesday–Dienstag`, and so on.

The value type of a dictionary can be any user-defined type. However, the key type of a dictionary is limited to one of the following types:

- Integral types (`byte`, `short`, `int`, `long`, `bool`, and enumerated types)
- `string`
- sequences with elements of integral type or `string`
- structures containing only data members of integral type or `string`

Complex nested types, such as nested structures or dictionaries, and floating-point types (`float` and `double`) cannot be used as the key type. Complex nested types are disallowed because they complicate the language mappings for dictionaries, and floating-point types are disallowed because representational changes of values as they cross machine boundaries can lead to ill-defined semantics for equality.

4.7.5 Constant Definitions and Literals

Slice allows you to define constants. Constant definitions must be of one of the following types:

- An integral type (`bool`, `byte`, `short`, `int`, `long`, or an enumerated type)
- `float` or `double`
- `string`

Here are a few examples:

```
const bool AppendByDefault = true;
const byte LowerNibble = 0x0f;
const string Advice = "Don't Panic!";
const short TheAnswer = 42;
const double PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

The syntax for literals is the same as for C++ and Java (with a few minor exceptions):

- Boolean constants can only be initialized with the keywords `false` and `true`. (You cannot use `0` and `1` to represent `false` and `true`.)
- As for C++, integer literals can be specified in decimal, octal, or hexadecimal notation. For example:

```
const byte TheAnswer = 42;
const byte TheAnswerInOctal = 052;
const byte TheAnswerInHex = 0x2A;           // or 0x2a
```

Be aware that, if you interpret `byte` as a number instead of a bit pattern, you will get different results in different languages. For example, in C++, `byte` maps to `char`, which may be signed or unsigned, depending on the target platform.

Note that suffixes to indicate long and unsigned constants (`l`, `L`, `u`, `U`, used by C++) are illegal:

```
const long Wrong = 0u;           // Syntax error
const long WrongToo = 1000000L; // Syntax error
```

The value of an integer literal must be within the range of its constant type, as shown in Table 4.1 on page 62; otherwise the compiler will issue a diagnostic.

- Floating-point literals use C++ syntax, except that you cannot use an `l` or `L` suffix to indicate an extended floating-point constant; however, `f` and `F` are legal (but are ignored). Here are a few examples:

```
const float P1 = -3.14f;      // Integer & fraction, with suffix
const float P2 = +3.1e-3;    // Integer, fraction, and exponent
const float P3 = .1;         // Fraction part only
const float P4 = 1.;         // Integer part only
const float P5 = .9E5;       // Fraction part and exponent
const float P6 = 5e2;        // Integer part and exponent
```

Floating-point literals must be within the range of the constant type (`float` or `double`); otherwise, the compiler will issue a diagnostic.

- String literals support the same escape sequences as C++. Here are some examples:

```
const string AnOrdinaryString = "Hello World!";

const string DoubleQuote =      "\"";
const string TwoSingleQuotes =  "'\\''";      // ' and \ are OK
const string Newline =          "\n";
const string CarriageReturn =   "\r";
const string HorizontalTab =   "\t";
const string VerticalTab =     "\v";
const string FormFeed =         "\f";
const string Alert =            "\a";
const string Backspace =        "\b";
const string QuestionMark =    "\?";
const string Backslash =         "\\\";

const string OctalEscape =      "\007";      // Same as \a
const string HexEscape =        "\x07";      // Ditto

const string UniversalCharName = "\u03A9"; // Greek Omega
```

As for C++, adjacent string literals are concatenated:

```
const string MSG1 = "Hello World!";
const string MSG2 = "Hello" " " "World!";           // Same message

/*
 * Escape sequences are processed before concatenation,
 * so the string below contains two characters,
 * '\xa' and 'c'.
 */

const string S = "\xa" "c";
```

Note that Slice has no concept of a null string:

```
const string nullString = 0;    // Illegal!
```

Null strings simply do not exist in Slice and, therefore, do not exist as a legal value for a string anywhere in the Ice platform. The reason for this decision is that null strings do not exist in many programming languages.⁵

4.8 Interfaces, Operations, and Exceptions

The central focus of Slice is on defining interfaces, for example:

```
struct TimeOfDay {
    short hour;          // 0 - 23
    short minute;        // 0 - 59
    short second;        // 0 - 59
};

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};
```

This definition defines an interface type called `Clock`. The interface supports two operations: `getTime` and `setTime`. Clients access an object supporting the `Clock` interface by invoking an operation on the proxy for the object: to read the current time, the client invokes the `getTime` operation; to set the current time, the client invokes the `setTime` operation, passing an argument of type `TimeOfDay`.

5. Many languages other than C and C++ use a byte array as the internal string representation. Null strings do not exist (and would be very difficult to map) in such languages.

Invoking an operation on a proxy instructs the Ice run time to send a message to the target object. The target object can be in another address space or can be collocated (in the same process) as the caller—the location of the target object is transparent to the client. If the target object is in another (possibly remote) address space, the Ice run time invokes the operation via a remote procedure call; if the target is collocated with the client, the Ice run time uses an ordinary function call instead, to avoid the overhead of marshaling.

You can think of an interface definition as the equivalent of the public part of a C++ class definition or as the equivalent of a Java interface, and of operation definitions as (virtual) member functions. Note that nothing but operation definitions are allowed to appear inside an interface definition. In particular, you cannot define a type, an exception, or a data member inside an interface. This does not mean that your object implementation cannot contain state—it can, but how that state is implemented (in the form of data members or otherwise) is hidden from the client and, therefore, need not appear in the object’s interface definition.

An Ice object has exactly one (most derived) Slice interface type (or class type—see Section 4.9). Of course, you can create multiple Ice objects that have the same type; to draw the analogy with C++, a Slice interface corresponds to a C++ class *definition*, whereas an Ice object corresponds to a C++ class *instance* (but Ice objects can be implemented in multiple different address spaces).

Ice also provides multiple interfaces via a feature called *facets*. We discuss facets in detail in Chapter 18.

A Slice interface defines the smallest grain of distribution in Ice: each Ice object has a unique identity (encapsulated in its proxy) that distinguishes it from all other Ice objects; for communication to take place, you must invoke operations on an object’s proxy. There is no other notion of an addressable entity in Ice. You cannot, for example, instantiate a Slice structure and have clients manipulate that structure remotely. To make the structure accessible, you must create an interface that allows clients to access the structure.

The partition of an application into interfaces therefore has profound influence on the overall architecture. Distribution boundaries must follow interface (or class) boundaries; you can spread the implementation of interfaces over multiple address spaces (and you can implement multiple interfaces in the same address space), but you cannot implement parts of interfaces in different address spaces.

4.8.1 Parameters and Return Values

An operation definition must contain a return type and zero or more parameter definitions. For example, the `getTime` operation on page 71 has a return type of `TimeOfDay` and the `setTime` operation has a return type of `void`. You must use `void` to indicate that an operation returns no value—there is no default return type for Slice operations.

An operation can have one or more input parameters. For example, `setTime` accepts a single input parameter of type `TimeOfDay` called `time`. Of course, you can use multiple input parameters, for example:

```
interface CircadianRhythm {  
    void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);  
    // ...  
};
```

Note that the parameter name (as for Java) is mandatory. You cannot omit the parameter name, so the following is in error:

```
interface CircadianRhythm {  
    void setSleepPeriod(TimeOfDay, TimeOfDay); // Error!  
    // ...  
};
```

By default, parameters are sent from the client to the server, that is, they are input parameters. To pass a value from the server to the client, you can use an output parameter, indicated by the `out` keyword. For example, an alternative way to define the `getTime` operation on page 71 would be:

```
void getTime(out TimeOfDay time);
```

This achieves the same thing but uses an output parameter instead of the return value. As with input parameters, you can use multiple output parameters:

```
interface CircadianRhythm {  
    void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);  
    void getSleepPeriod(out TimeOfDay startTime,  
                      out TimeOfDay stopTime);  
    // ...  
};
```

If you have both input and output parameters for an operation, the output parameters must follow the input parameters:

```

void changeSleepPeriod(    TimeOfDay startTime,      // OK
                           TimeOfDay stopTime,
                           out TimeOfDay prevStartTime,
                           out TimeOfDay prevStopTime);
void changeSleepPeriod(out TimeOfDay prevStartTime,
                      out TimeOfDay prevStopTime,
                      TimeOfDay startTime,      // Error
                      TimeOfDay stopTime);

```

Slice does not support parameters that are both input and output parameters (call by reference). The reason is that, for remote calls, reference parameters do not result in the same savings that one can obtain for call by reference in programming languages. (Data still needs to be copied in both directions and any gains in marshaling efficiency are negligible.) Also, reference (or input–output) parameters result in more complex language mappings, with concomitant increases in code size.

Style of Operation Definition

As you would expect, language mappings follow the style of operation definition you use in Slice: Slice return types map to programming language return types, and Slice parameters map to programming language parameters.

For operations that return only a single value, it is common to return the value from the operation instead of using an `out`-parameter. This style maps naturally into all programming languages. Note that, if you use an `out`-parameter instead, you impose a different API style on the client: most programming languages permit the return value of a function to be ignored whereas it is typically not possible to ignore an output parameter.

For operations that return multiple values, it is common to return all values as `out`-parameters and to use a return type of `void`. However, the rule is not all that clear-cut because operations with multiple output values can have one particular value that is considered more “important” than the remainder. A common example of this is an iterator operation that returns items from a collection one-by-one:

```
bool next(out RecordType r);
```

The `next` operation returns two values: the record that was retrieved and a Boolean to indicate the end-of-collection condition. (If the return value is `false`, the end of the collection has been reached and the parameter `r` has an undefined value.) This style of definition can be useful because it naturally fits into the way programmers write control structures. For example:

```
while (next(record))
    // Process record...

if (next(record))
    // Got a valid record...
```

Overloading

Slice does not support any form of overloading of operations. For example:

```
interface CircadianRhythm {
    void modify(TimeOfDay startTime,
                TimeOfDay endTime);
    void modify(    TimeOfDay startTime,           // Error
                  TimeOfDay endTime,
                  out TimeOfDay prevStartTime,
                  out TimeOfDay prevEndTime);
};
```

Operations in the same interface must have different names, regardless of what type and number of parameters they have. This restriction exists because overloaded functions cannot sensibly be mapped to languages without built-in support for overloading.⁶

Nonmutating Operations

Some operations, such as `getTime` on page 71, do not modify the state of the object they operate on. They are the conceptual equivalent of C++ `const` member functions. You can indicate this in Slice as follows:

```
interface Clock {
    non mutating TimeOfDay getTime();
    void setTime(TimeOfDay time);
};
```

The `non mutating` keyword indicates that the `getTime` operation does not change the state of its object. This is useful for two reasons:

- Language mappings can take advantage of the additional knowledge about the behavior of the operation. For example, for C++, `non mutating` operations map to C++ `const` member functions on skeleton classes.

6. Name mangling is not an option in this case: while it works fine for compilers, it is unacceptable to humans.

- Knowing that an operation will not modify the state of its object permits the Ice run time to attempt more aggressive error recovery. Specifically, Ice guarantees *at-most-once* semantics for operation invocations.

For normal operations, the Ice run time has to be conservative about how it deals with errors. For example, if a client sends an operation invocation to a server and then loses connectivity, there is no way for the client-side run time to find out whether the request it sent actually made it to the server. This means that the run time cannot attempt to recover from the error by re-establishing a connection and sending the request a second time because that could cause the operation to be invoked a second time and violate at-most-once semantics; the run time has no option but to report the error to the application.

For **nonmutating** operations, on the other hand, the client-side run time can attempt to re-establish a connection to the server and safely send the failed request a second time. If the server can be reached on the second attempt, everything is fine and the application never notices the (temporary) failure. Only if the second attempt fails need the run time report the error back to the application. (The number of retries can be increased with an Ice configuration parameter.)

Idempotent Operations

We can further modify the definition of the `Clock` interface on page 75 to indicate that the `setTime` operation is *idempotent*:

```
interface Clock {  
    nonmutating TimeOfDay getTime();  
    idempotent void setTime(TimeOfDay time);  
};
```

An operation is **idempotent** if two successive invocations of the operation have the same effect as a single invocation. For example, `x = 1;` is an **idempotent** operation because it does not matter whether it is executed once or twice—either way, `x` ends up with the value 1. On the other hand, `x += 1;` is not an **idempotent** operation because executing it twice results in a different value for `x` than executing it once.

The `idempotent` keyword indicates that an operation can safely be executed more than once. As for **nonmutating** operations, the Ice run time uses this knowledge to recover more aggressively from errors.

An operation can have either a **nonmutating** or an **idempotent** qualifier, but not both. (**nonmutating** implies **idempotent**.)

4.8.2 User Exceptions

Looking at the `setTime` operation on page 71, we find a potential problem: given that the `TimeOfDay` structure uses `short` as the type of each field, what will happen if a client invokes the `setTime` operation and passes a `TimeOfDay` value with meaningless field values, such as -199 for the minute field, or 42 for the hour? Obviously, it would be nice to provide some indication to the caller that this is meaningless. Slice allows you to define user exceptions to indicate error conditions to the client. For example:

```
exception Error {} // Empty exceptions are legal

exception RangeError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};
```

A user exception is much like a structure in that it contains a number of data members. However, unlike structures, exceptions can have zero data members, that is, be empty. Exceptions allow you to return an arbitrary amount of error information to the client if an error condition arises in the implementation of an operation. Operations use an exception specification to indicate the exceptions that may be returned to the client:

```
interface Clock {
    nonmutating TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time)
        throws RangeError, Error;
};
```

This definition indicates that the `setTime` operation may throw either a `RangeError` or an `Error` user exception (and no other type of exception). If the client receives a `RangeError` exception, the exception contains the `TimeOfDay` value that was passed to `setTime` and caused the error (in the `errorTime` member), as well as the minimum and maximum time values that can be used (in the `minTime` and `maxTime` members). If `setTime` failed because of an error not caused by an illegal parameter value, it throws `Error`. Obviously, because `Error` does not have data members, the client will have no idea what exactly it was that went wrong—it simply knows that the operation did not work.

An operation can throw only those user exceptions that are listed in its exception specification. If, at run time, the implementation of an operation throws an exception that is not listed in its exception specification, the client receives a run-

time exception (see Section 4.8.4) to indicate that the operation did something illegal. To indicate that an operation does not throw any user exception, simply omit the exception specification. (There is no empty exception specification in Slice.)

Exceptions are not first-class data types and first-class data types are not exceptions:

- You cannot pass an exception as a parameter value.
- You cannot use an exception as the type of a data member.
- You cannot use an exception as the element type of a sequence.
- You cannot use an exception as the key or value type of a dictionary.
- You cannot throw a value of non-exception type (such as a value of type `int` or `string`).

The reason for these restrictions is that some implementation languages use a specific and separate type for exceptions (in the same way as Slice does). For such languages, it would be difficult to map exceptions if they could be used as an ordinary data type. (C++ is somewhat unusual among programming languages by allowing arbitrary types to be used as exceptions.)

4.8.3 Exception Inheritance

Exceptions support inheritance. For example:

```
exception ErrorBase {
    string reason;
};

enum RTError {
    DivideByZero, NegativeRoot, IllegalNull /* ... */
};

exception RuntimeError extends ErrorBase {
    RTError err;
};

enum LError { ValueOutOfRange, ValuesInconsistent, /* ... */ };

exception LogicError extends ErrorBase {
    LError err;
};

exception RangeError extends LogicError {
```

```
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};
```

These definitions set up a simple exception hierarchy:

- `ErrorBase` is at the root of the tree and contains a string explaining the cause of the error.
- Derived from `ErrorBase` are `RuntimeError` and `LogicError`. Each of these exceptions contains an enumerated value that further categorizes the error.
- Finally, `RangeError` is derived from `LogicError` and reports the details of the specific error.

Setting up exception hierarchies such as this not only helps to create a more readable specification because errors are categorized, but also can be used at the language level to good advantage. For example, the Slice C++ mapping preserves the exception hierarchy so you can catch exceptions generically as a base exception, or set up exception handlers to deal with specific exceptions.

Looking at the exception hierarchy on page 78, it is not clear whether, at run time, the application will only throw most derived exceptions, such as `RangeError`, or if it will also throw base exceptions, such as `LogicError`, `RuntimeError`, and `ErrorBase`. If you want to indicate that a base exception, interface, or class is abstract (will not be instantiated), you can add a comment to that effect.

Note that, if the exception specification of an operation indicates a specific exception type, at run time, the implementation of the operation may also throw more derived exceptions. For example:

```
exception Base {
    // ...
};

exception Derived extends Base {
    // ...
};

interface Example {
    void op() throws Base;      // May throw Base or Derived
};
```

In this example, `op` may throw a `Base` or a `Derived` exception, that is, any exception that is compatible with the exception types listed in the exception specification can be thrown at run time.

As a system evolves, it is quite common for new, derived exceptions to be added to an existing hierarchy. Assume that we initially construct clients and server with the following definitions:

```
exception Error {  
    // ...  
};  
  
interface Application {  
    void doSomething() throws Error;  
};
```

Also assume that a large number of clients are deployed in field, that is, when you upgrade the system, you cannot easily upgrade all the clients. As the application evolves, a new exception is added to the system and the server is redeployed with the new definition:

```
exception Error {  
    // ...  
};  
  
exception FatalApplicationError extends Error {  
    // ...  
};  
  
interface Application {  
    void doSomething() throws Error;  
};
```

This raises the question of what should happen if the server throws a `FatalApplicationError` from `doSomething`. The answer depends whether the client was built using the old or the updated definition:

- If the client was built using the same definition as the server, it simply receives a `FatalApplicationError`.
- If the client was built with the original definition, that client has no knowledge that `FatalApplicationError` even exists. In this case, the Ice run time automatically slices the exception to the most-derived type that is understood by the receiver (`Error`, in this case) and discards the information that is specific to the derived part of the exception. (This is exactly analogous to catching C++ exceptions by value—the exception is sliced to the type used in the `catch`-clause.)

Exceptions support single inheritance only. (Multiple inheritance would be difficult to map into many programming languages.)

4.8.4 Ice Run-Time Exceptions

As mentioned in Section 2.2.2, in addition to any user exceptions that are listed in an operation's exception specification, an operation can also throw Ice *run-time exceptions*. Run-time exceptions are predefined exceptions that indicate platform-related run-time errors. For example, if a networking error interrupts communication between client and server, the client is informed of this by a run-time exception, such as `ConnectTimeoutException` or `SocketException`.

The exception specification of an operation must not list any run-time exceptions. (It is understood that all operations can raise run-time exceptions and you are not allowed to restate that.)

Inheritance Hierarchy for Exceptions

All the Ice run-time and user exceptions are arranged in an inheritance hierarchy, as shown in Figure 4.3.

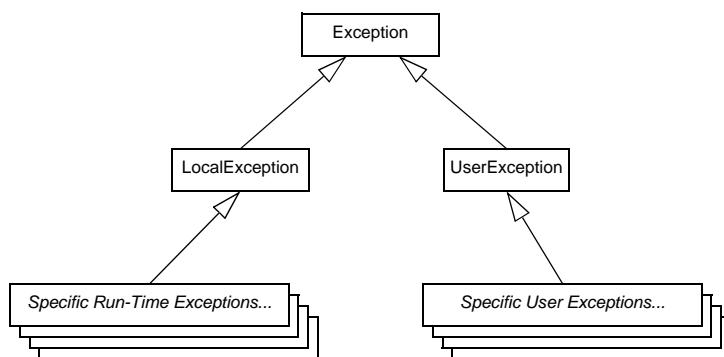


Figure 4.3. Inheritance structure for Ice run-time exceptions.

`Ice::Exception` is at the root of the inheritance hierarchy. Derived from that are the (abstract) types `Ice::LocalException` and `Ice::UserException`. In turn, all run-time exceptions are derived from `Ice::LocalException`, and all user exceptions are derived from `Ice::UserException`.

Figure 4.4 shows the complete hierarchy of the Ice run-time exceptions.⁷

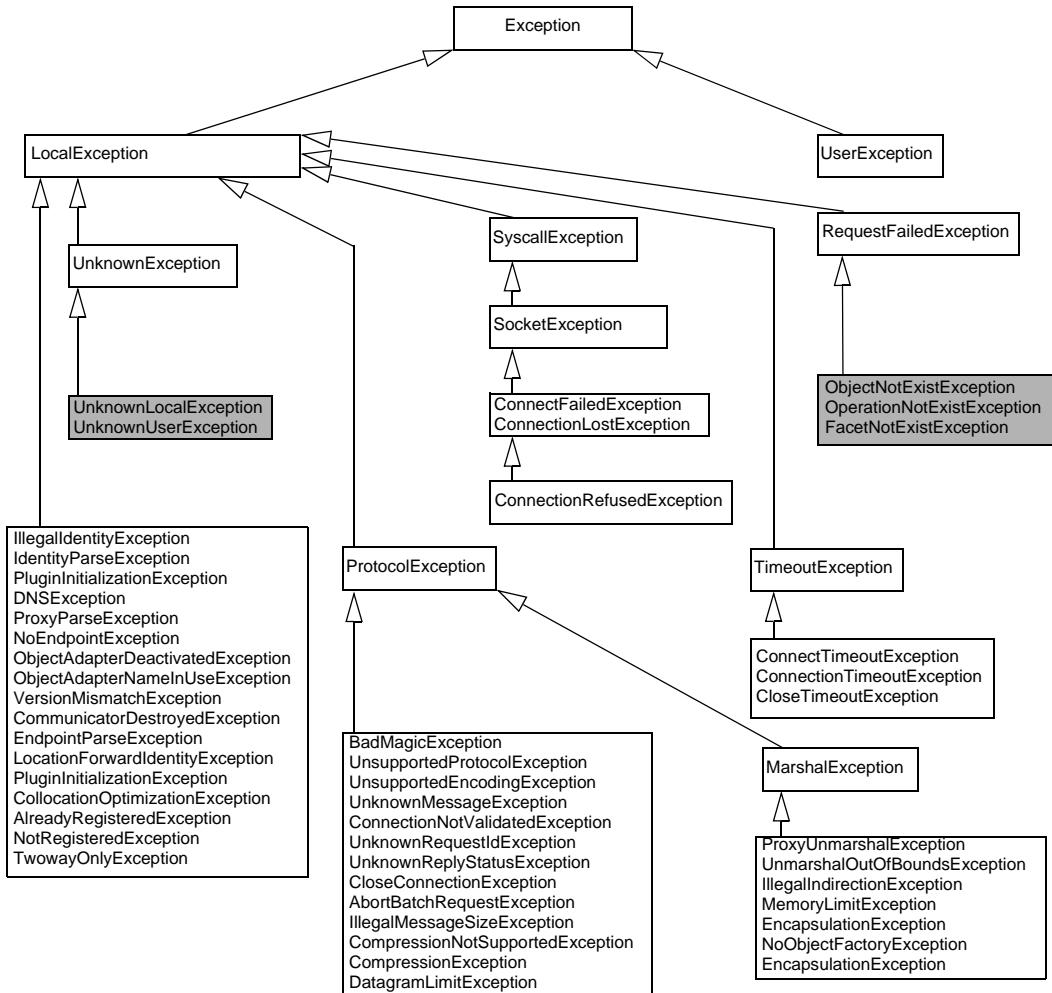


Figure 4.4. Ice run-time exception hierarchy. (Shaded exceptions can be sent by the server.)

7. We use the Unified Modeling Language (UML) for the object model diagrams in this book (see [1] and [13] for details).

Note that Figure 4.4 groups several exceptions into a single box to save space (which, strictly, is incorrect UML syntax). Also note that some run-time exceptions have data members, which, for brevity, we have omitted in Figure 4.4. These data members provide additional information about the precise cause of an error.

Many of the run-time exceptions have self-explanatory names, such as `MemoryLimitException`. Others indicate problems in the Ice run time, such as `EncapsulationException`. Still others can arise only through application programming errors, such as `NotRegisteredException`. In practice, you will likely never see most of these exceptions. However, there are a few run-time exceptions you will encounter and whose meaning you should know.

Local Versus Remote Exceptions

Most error conditions are detected on the client side. For example, if an attempt to contact a server fails, the client-side run time raises a `ConnectTimeoutException`. However, there are three specific error conditions (shaded in Figure 4.4) that are detected by the server and made known explicitly to the client-side run time via the Ice protocol:

- `ObjectNotExistException`

This exception indicates that a request was delivered to the server but the server could not locate a servant with the identity that is embedded in the proxy. In other words, the server could not find an object to dispatch the request to.

An `ObjectNotExistException` is a death certificate: it indicates that the target object in the server does not exist and, moreover, will never exist.⁸ If you receive this exception, you are expected to clean up whatever resources you might have allocated that relate to the specific object for which you receive this exception.

- `FacetNotExistException`

The client attempted to contact a non-existent facet of an object, that is, the server has at least one servant with the given identity, but no servant with a matching facet name. (See Chapter 18 for a discussion of facets.)

8. The Ice run time raises `ObjectNotExistException` only if there are no facets in existence with a matching identity; otherwise, it raises `FacetNotExistException` (see Chapter 18).

- **OperationNotExistException**

This exception is raised if the server could locate an object with the correct identity but, on attempting to dispatch the client's operation invocation, the server found that the target object does not have such an operation. You will see this exception in only two cases:

- You have used an unchecked down-cast on a proxy of the incorrect type. (See page 160 and page 212 for unchecked down-casts.)
- Client and server have been built with Slice definitions for an interface that disagree with each other, that is, the client was built with an interface definition for the object that indicates that an operation exists, but the server was built with a different version of the interface definition in which the operation is absent.

Any error condition on the server side that is not described by one of the three preceding exceptions is made known to the client as one of two generic exceptions (shaded in Figure 4.4):

- **UnknownUserException**

This exception indicates that an operation implementation has thrown an exception that is not declared in the operation's exception specification. For example, if the operation in the server throws a C++ exception, such as a `char *`, or a Java exception, such as a `ClassCastException`, the client receives an `UnknownUserException`.

- **UnknownLocalException**

If an operation implementation raises a run-time exception other than `ObjectNotExistException`, `FacetNotExistException`, or `OperationNotExistException` (such as a `NotRegisteredException`), the client receives an `UnknownLocalException`. In other words, the Ice protocol does not transmit the exact exception that was encountered in the server, but simply returns a bit to the client in the reply to indicate that the server encountered a run-time exception.

A common cause for a client receiving an `UnknownLocalException` is failure to catch and handle all exceptions in the server. For example, if the implementation of an operation encounters an exception it does not handle, the exception propagates all the way up the call stack until the stack is unwound to the point where the Ice run time invoked the operation. The Ice run time catches all exceptions that "escape" from an operation invocation and returns them to the client as an `UnknownLocalException`.

All other run-time exceptions (not shaded in Figure 4.4) are detected by the client-side run time and are raised locally.

It is possible for the implementation of an operation to throw Ice run-time exceptions (as well as user exceptions). For example, if a client holds a proxy to an object that no longer exists in the server, your server application code is required to throw an `ObjectNotExistException`. If you do throw run-time exceptions from your application code, you should take care to throw a run-time exception only if appropriate, that is, do not use run-time exceptions to indicate something that really should be a user exception. Doing so can be very confusing to the client: if the application “hijacks” some run-time exceptions for its own purposes, the client can no longer decide whether the exception was thrown by the Ice run time or by the server application code. This can make debugging very difficult.

4.8.5 Interface Semantics and Proxies

Building on the `Clock` example, we can create definitions for a world-time server:

```
exception GenericError {
    string reason;
};

struct TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};

exception BadTimeVal extends GenericError {};

interface Clock {
    nonmutating TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time) throws BadTimeVal;
};

dictionary<string, Clock*> TimeMap; // Time zone name to clock map

exception BadZoneName extends GenericError {};

interface WorldTime {
    idempotent void addZone(string zoneName, Clock* zoneClock);
    void removeZone(string zoneName) throws BadZoneName;
    nonmutating Clock* findZone(string zoneName)
```

```

    throws BadZoneName;
nonmutating TimeMap listZones();
idempotent void setZones(TimeMap zones);
};
```

The `WorldTime` interface acts as a collection manager for clocks, one for each time zone. In other words, the `WorldTime` interface manages a collection of pairs. The first member of each pair is a time zone name; the second member of the pair is the clock that provides the time for that zone. The interface contains operations that permit you to add or remove a clock from the map (`addZone` and `removeZone`), to search for a particular time zone by name (`findZone`), and to read or write the entire map (`listZones` and `setZones`).

The `WorldTime` example illustrates an important Slice concept: note that `addZone` accepts a parameter of type `Clock*` and `findZone` returns a parameter of type `Clock*`. In other words, interfaces are types in their own right and can be passed as parameters. The `*` operator is known as the *proxy operator*. Its left-hand argument must be an interface (or class—see Section 4.9) and its return type is a proxy. A proxy is like a pointer that can denote an object. The semantics of proxies are very much like those of C++ class instance pointers:

- A proxy can be null (see page 91).
- A proxy can dangle (point at an object that is no longer there)
- Operations dispatched via a proxy use late binding: if the actual run-time type of the object denoted by the proxy is more derived than the proxy's type, the implementation of the most-derived interface will be invoked.

When a client passes a `Clock` proxy to the `addZone` operation, the proxy denotes an actual `Clock` object in a server. The *Clock Ice object* denoted by that proxy may be implemented in the same server process as the `WorldTime` interface, or in a different server process. Where the `Clock` object is physically implemented matters neither to the client nor to the server implementing the `WorldTime` interface; if either invokes an operation on a particular clock, such as `getTime`, an RPC call is sent to whatever server implements that particular clock. In other words, a proxy acts as a local “ambassador” for the remote object; invoking an operation on the proxy forwards the invocation to the actual object implementation. If the object implementation is in a different address space, this results in a remote procedure call; if the object implementation is collocated in the same address space, the Ice run time uses an ordinary local function call from the proxy to the object implementation.

Note that proxies also act very much like pointers in their sharing semantics: if two clients have a proxy to the same object, a state change made by one client (such as setting the time) will be visible to the other client.

Proxies are strongly typed (at least for statically typed languages, such as C++ and Java). This means that you cannot pass something other than a `Clock` proxy to the `addZone` operation; attempts to do so are rejected at compile time.

4.8.6 Interface Inheritance

Interfaces support inheritance. For example, we could extend our world-time server to support the concept of an alarm clock:

```
interface AlarmClock extends Clock {  
    nonmutating TimeOfDay getAlarmTime();  
    idempotent void      setAlarmTime(TimeOfDay alarmTime)  
                           throws BadTimeVal;  
};
```

The semantics of this are the same as for C++ or Java: `AlarmClock` is a subtype of `Clock` and an `AlarmClock` proxy can be substituted wherever a `Clock` proxy is expected. Obviously, an `AlarmClock` supports the same `getTime` and `setTime` operations as a `Clock` but also supports the `getAlarmTime` and `setAlarmTime` operations.

Multiple interface inheritance is also possible. For example, we can construct a radio alarm clock as follows:

```
interface Radio {  
    void setFrequency(long hertz) throws GenericError;  
    void setVolume(long dB) throws GenericError;  
};  
  
enum AlarmMode { RadioAlarm, BeepAlarm };  
  
interface RadioClock extends Radio, AlarmClock {  
    void      setMode(AlarmMode mode);  
    AlarmMode getMode();  
};
```

`RadioClock` extends both `Radio` and `AlarmClock` and can therefore be passed where a `Radio`, an `AlarmClock`, or a `Clock` is expected. The inheritance diagram for this definition looks as follows:

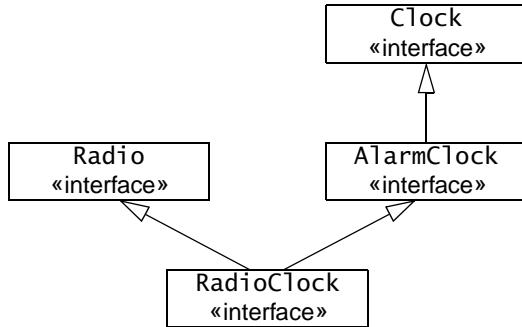


Figure 4.5. Inheritance diagram for `RadioClock`.

Interfaces that inherit from more than one base interface may share a common base interface. For example, the following definition is legal:

```

interface B { /* ... */ };
interface I1 extends B { /* ... */ };
interface I2 extends B { /* ... */ };
interface D extends I1, I2 { /* ... */ };
  
```

This definition results in the familiar diamond shape:

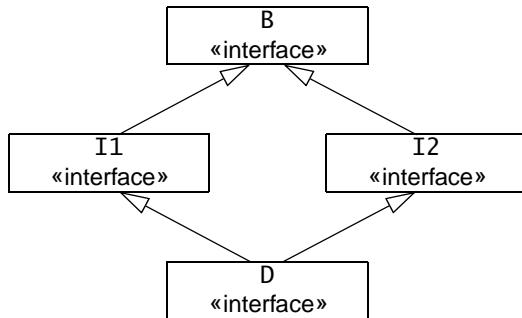


Figure 4.6. Diamond-shaped inheritance.

Interface Inheritance Limitations

If an interface uses multiple inheritance, it must not inherit the same operation name from more than one base interface. For example, the following definition is illegal:

```
interface Clock {
    void set(TimeOfDay time);                                // set time
};

interface Radio {
    void set(long hertz);                                    // set frequency
};

interface RadioClock extends Radio, Clock {      // Illegal!
    // ...
};
```

This definition is illegal because `RadioClock` inherits two `set` operations, `Radio::set` and `Clock::set`. The Slice compiler makes this illegal because (unlike C++) many programming languages have built-in facility for disambiguating the different operations. In Slice, the simple rule is that all inherited operations must have unique names. (In practice, this is rarely a problem because inheritance is rarely added to an interface hierarchy “after the fact”. To avoid accidental clashes, we suggest that you use descriptive operation names, such as `setTime` and `setFrequency`. This makes accidental name clashes less likely.)

Implicit Inheritance from Object

All Slice interfaces are ultimately derived from `Object`. For example, the inheritance hierarchy from Figure 4.5 would be shown more correctly as in Figure 4.7.

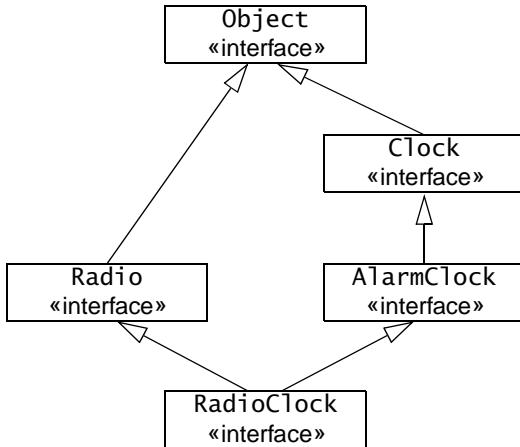


Figure 4.7. Implicit inheritance from `Object`.

Because all interfaces have a common base interface, we can pass any type of interface as that type. For example:

```
interface ProxyStore {
    idempotent void putProxy(string name, Object* o);
    nonmutating Object* getProxy(string name);
};
```

`Object` is a Slice keyword (note the capitalization) that denotes the root type of the inheritance hierarchy. The `ProxyStore` interface is a generic proxy storage facility: the client can call `putProxy` to add a proxy of any type under a given name and later retrieve that proxy again by calling `getProxy` and supplying that name. The ability to generically store proxies in this fashion allows us to build general-purpose facilities, such as a naming service that can store proxies and deliver them to clients. Such a service, in turn, allows us to avoid hard-coding proxy details into clients and servers (see Chapter 21).

Inheritance from type `Object` is always implicit. For example, the following Slice definition is illegal:

```
interface MyInterface extends Object { /* ... */ }; // Error!
```

It is understood that all interfaces inherit from type `Object`; you are not allowed to restate that.

Null Proxies

Looking at the `ProxyStore` interface once more, we notice that `getProxy` does not have an exception specification. The question then is what should happen if a client calls `getProxy` with a name under which no proxy is stored? Obviously, we could add an exception to indicate this condition to `getProxy`. However, another option is to return a *null proxy*. Ice has the built-in notion of a null proxy, which is a proxy that “points nowhere”. When such a proxy is returned to the client, the client can test the value of the returned proxy to check whether it is null or denotes a valid object.

A more interesting question is: “which approach is more appropriate, throwing an exception or returning a null proxy?” The answer depends on the expected usage pattern of an interface. For example, if, in normal operation, you do not expect clients to call `getProxy` with a non-existent name, it is better to throw an exception. (This is probably the case for our `ProxyStore` interface: the fact that there is no `list` operation makes it clear that clients are expected to know which names are in use.)

On the other hand, if you expect that clients will occasionally try to look up something that is not there, it is better to return a null proxy. The reason is that throwing an exception breaks the normal flow of control in the client and requires special handling code. This means that you should throw exceptions only in exceptional circumstances. For example, throwing an exception if a database lookup returns an empty result set is wrong; it is expected and normal that a result set is occasionally empty.

It is worth paying attention to such design issues: well-designed interfaces that get these details right are easier to use and easier to understand. Not only do such interfaces make life easier for client developers, they also make it less likely that latent bugs cause problems later.

Self-Referential Interfaces

Proxies have pointer semantics, so we can define self-referential interfaces. For example:

```
interface Link {
    nonmutating SomeType getValue();
    nonmutating Link* next();
};
```

The `Link` interface contains a `next` operation that returns a proxy to a `Link` interface. Obviously, this can be used to create a chain of interfaces; the final link in the chain returns a null proxy from its `next` operation.

Empty Interfaces

The following Slice definition is legal:

```
interface Empty {};
```

The Slice compiler will compile this definition without complaint. An interesting question is: “why would I need an empty interface?” In most cases, empty interfaces are an indication of design errors. Here is one example:

```
interface ThingBase {};  
  
interface Thing1 extends ThingBase {  
    // Operations here...  
};  
  
interface Thing2 extends ThingBase {  
    // Operations here...  
};
```

Looking at this definition, we can make two observations:

- `Thing1` and `Thing2` have a common base and are therefore related.
- Whatever is common to `Thing1` and `Thing2` can be found in interface `ThingBase`.

Of course, looking at `ThingBase`, we find that `Thing1` and `Thing2` do not share any operations at all because `ThingBase` is empty. Given that we are using an object-oriented paradigm, this is definitely strange: in the object-oriented model, the *only* way to communicate with an object is to send a message to the object. But, to send a message, we need an operation. Given that `ThingBase` has no operations, we cannot send a message to it, and it follows that `Thing1` and `Thing2` are *not* related because they have no common operations. But of course, seeing that `Thing1` and `Thing2` have a common base, we conclude that they *are* related, otherwise the common base would not exist. At this point, most programmers begin to scratch their head and wonder what is going on here.

One common use of the above design is a desire to treat `Thing1` and `Thing2` polymorphically. For example, we might continue the previous definition as follows:

```
interface ThingUser {
    void putThing(ThingBase* thing);
};
```

Now the purpose of having the common base becomes clear: we want to be able to pass both `Thing1` and `Thing2` proxies to `putThing`. Does this justify the empty base interface? To answer this question, we need to think about what happens in the implementation of `putThing`. Obviously, `putThing` cannot possibly invoke an operation on a `ThingBase` because there are no operations. This means that `putThing` must do one of two things:

1. `putThing` can simply remember the value of `thing`.
2. `putThing` can try to down-cast to either `Thing1` or `Thing2` and then invoke an operation. The pseudo-code for the implementation of `putThing` would look something like this:

```
void putThing(ThingBase thing)
{
    if (is_a(Thing1, thing)) {
        // Do something with Thing1...
    } else if (is_a(Thing2, thing)) {
        // Do something with Thing2...
    } else {
        // Might be a ThingBase?
        // ...
    }
}
```

The implementation tries to down-cast its argument to each possible type in turn until it has found the actual run-time type of the argument. Of course, any object-oriented text book worth its price will tell you that this is an abuse of inheritance and leads to maintenance problems.

If you find yourself writing operations such as `putThing` that rely on artificial base interfaces, ask yourself whether you really need to do things this way. For example, a more appropriate design might be:

```
interface Thing1 {
    // Operations here...
};

interface Thing2 {
    // Operations here...
};
```

```
interface ThingUser {
    void putThing1(Thing1* thing);
    void putThing2(Thing2* thing);
};
```

With this design, `Thing1` and `Thing2` are not related, and `ThingUser` offers a separate operation for each type of proxy. The implementation of these operations does not need to use any down-casts, and all is well in our object-oriented world.

Another common use of empty base interfaces is the following:

```
interface PersistentObject {};

interface Thing1 extends PersistentObject {
    // Operations here...
};

interface Thing2 extends PersistentObject {
    // Operations here...
};
```

Clearly, this design places persistence functionality into the `PersistentObject` base and requires objects that want to have persistent state to inherit from `PersistentObject`. On the face of things, this is reasonable: after all, using inheritance in this way is a well-established design pattern, so what can possibly be wrong with it? As it turns out, there are a number of things that are wrong with this design:

- The above inheritance hierarchy is used to add *behavior* to `Thing1` and `Thing2`. However, in a strict OO model, behavior can be invoked only by sending messages.

This raises the question of how `PersistentObject` actually goes about doing its job; presumably, it knows something about the implementation (that is, the internal state) of `Thing1` and `Thing2`, so it can write that state into a database. But, if so, `PersistentObject`, `Thing1`, and `Thing2` can no longer be implemented in different address spaces because, in that case, `PersistentObject` can no longer get at the state of `Thing1` and `Thing2`.

Alternatively, `Thing1` and `Thing2` use some functionality provided by `PersistentObject` in order to make their internal state persistent. But `PersistentObject` does not have any operations, so how would `Thing1` and `Thing2` actually go about achieving this? Again, the only way that can work is if `PersistentObject`, `Thing1`, and `Thing2` are implemented in a single

address space and share implementation state behind the scenes, meaning that they cannot be implemented in different address spaces.

- The above inheritance hierarchy splits the world into two halves, one containing persistent objects and one containing non-persistent ones. This has far-reaching ramifications:
 - Suppose you have an existing application with already implemented, non-persistent objects. Requirements change over time and you find that you now would like to make some of your objects persistent. With the above design, you cannot do this unless you change the type of your objects because they now must inherit from `PersistentObject`. Of course, this is extremely bad news: not only do you have to change the implementation of your objects in the server, you also need to locate and update all the clients that are currently using your objects because they suddenly have a completely new type. What is worse, there is no way to keep things backward compatible: either all clients change with the server, or none of them do. It is impossible for some clients to remain “unupgraded”.
 - The design does not scale to multiple features. Imagine that we have a number of additional behaviors that objects can inherit, such as serialization, fault-tolerance, persistence, and the ability to be searched by a search engine. We quickly end up in a mess of multiple inheritance. What is worse, each possible combination of features creates a completely separate type hierarchy. This means that you can no longer write operations that generically operate on a number of object types. For example, you cannot pass a persistent object to something that expects a non-persistent object, *even if the receiver of the object does not care about the persistence aspects of the object*. This quickly leads to fragmented and hard-to-maintain type systems. Before long, you will either find yourself rewriting your application or end up with something that is both difficult to use and difficult to maintain.

The foregoing discussion will hopefully serve as a warning: Slice is an *interface* definition language that has nothing to do with *implementation* (but empty interfaces almost always indicate that implementation state is shared via mechanisms other than defined interfaces). If you find yourself writing an empty interface definition, at least step back and think about the problem at hand; there may be a more appropriate design that expresses your intent more cleanly. If you do decide to go ahead with an empty interface regardless, be aware that, almost certainly, you will lose the ability to later change the distribution of the object model over physical server process because you cannot place an address space boundary between interfaces that share hidden state.

Interface Versus Implementation Inheritance

Keep in mind that Slice interface inheritance applies only to *interfaces*. In particular, if two interfaces are in an inheritance relationship, this in no way implies that the implementations of those interfaces must also inherit from each other. You can choose to use implementation inheritance when you implement your interfaces, but you can also make the implementations independent of each other. (To C++ programmers, this often comes as a surprise because C++ uses implementation inheritance by default, and interface inheritance requires extra effort to implement.)

In summary, Slice inheritance simply establishes type compatibility. It says nothing about how interfaces are implemented and, therefore, keeps implementation choices open to whatever is most appropriate for your application.

4.9 Classes

In addition to interfaces, Slice permits the definition of classes. Classes are like interfaces in that they can have operations and are like structures in that they can have data members. This leads to hybrid objects that can be treated as interfaces and passed by reference, or can be treated as values and passed by value. Classes provide much architectural flexibility. For example, classes allow behavior to be implemented on the client side, whereas interfaces allow behavior to be implemented only on the server side.

Classes support inheritance and are therefore polymorphic: at run time, you can pass a class instance to an operation as long as the actual class type is derived from the formal parameter type in the operation's signature. This also permits classes to be used as type-safe unions, similarly to Pascal's discriminated variant records.

4.9.1 Simple Classes

A Slice class definition is similar to a structure definition, but uses the `class` keyword. For example:

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};
```

Apart from the keyword `class`, this definition is identical to the structure definition we saw on page 65. You can use a Slice class wherever you can use a Slice structure (but, as we will see shortly, for performance reasons, you should not use a class where a structure is sufficient). Unlike structures, classes can be empty:

```
class EmptyClass {}; // OK
struct EmptyStruct {}; // Error
```

Much the same design considerations as for empty interfaces (see page 92) apply to empty classes: you should at least stop and rethink your approach before committing yourself to an empty class.

4.9.2 Class Inheritance

Unlike structures, classes support inheritance. For example:

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};

class DateTime extends TimeOfDay {
    short day;            // 1 - 31
    short month;          // 1 - 12
    short year;           // 1753 onwards
};
```

This example illustrates one major reason for using a class: a class can be extended by inheritance, whereas a structure is not extensible. The previous example defines `DateTime` to extend the `TimeOfDay` class with a date.⁹

Classes only support single inheritance. The following is illegal:

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};
```

9. If you are puzzled by the comment about the year 1753, search the Web for “1752 date change”. The intricacies of calendars for various countries prior to that year can keep you occupied for months...

```

class Date {
    short day;
    short month;
    short year;
};

class DateTime extends TimeOfDay, Date { // Error!
    // ...
};

```

A derived class also cannot redefine a data member of its base class:

```

class Base {
    int integer;
};

class Derived extends Base {
    int integer;           // Error, integer redefined
};

```

4.9.3 Class Inheritance Semantics

Classes use the same pass-by-value semantics as structures. If you pass a class instance to an operation, the class and all its members are passed. The usual type compatibility rules apply: you can pass a derived instance where a base instance is expected. If the receiver has static type knowledge of the actual derived run-time type, it receives the derived instance; otherwise, if the receiver does not have static type knowledge of the derived type, the instance is sliced to the base type. For an example, suppose we have the following definitions:

```

// In file Clock.ice:

class TimeOfDay {
    short hour;          // 0 - 23
    short minute;        // 0 - 59
    short second;        // 0 - 59
};

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};

// In file DateTime.ice:

```

```
#include <Clock.ice>

class DateTime extends TimeOfDay {
    short day;           // 1 - 31
    short month;         // 1 - 12
    short year;          // 1753 onwards
};
```

Because `DateTime` is a sub-class of `TimeOfDay`, the server can return a `DateTime` instance from `getTime`, and the client can pass a `DateTime` instance to `setTime`. In this case, if both client and server are linked to include the code generated for both `Clock.ice` and `DateTime.ice`, they each receive the actual derived `DateTime` instance, that is, the actual run-time type of the instance is preserved.

Contrast this with the case where the server is linked to include the code generated for both `Clock.ice` and `DateTime.ice`, but the client is linked only with the code generated for `Clock.ice`. In other words, the server understands the type `DateTime` and can return a `DateTime` instance from `getTime`, but the client only understands `TimeOfDay`. In this case, the derived `DateTime` instance returned by the server is sliced to its `TimeOfDay` base type in the client. (The information in the derived part of the instance is simply lost to the client.)

Class hierarchies are useful if you need polymorphic *values* (instead of polymorphic *interfaces*). For example:

```
class Shape {
    // Definitions for shapes, such as size, center, etc.
};

class Circle extends Shape {
    // Definitions for circles, such as radius...
};

class Rectangle extends Shape {
    // Definitions for rectangles, such as width and length...
};

sequence<Shape> ShapeSeq;

interface ShapeProcessor {
    void processShapes(ShapeSeq ss);
};
```

Note the definition of `ShapeSeq` and its use as a parameter to the `processShapes` operation: the class hierarchy allows us to pass a polymorphic sequence of shapes (instead of having to define a separate operation for each type of shape).

The receiver of a `ShapeSeq` can iterate over the elements of the sequence and down-cast each element to its actual run-time type. (The receiver can also ask each element for its type ID to determine its type—see Section 6.14.1 and Section 8.11.2.)

4.9.4 Classes as Unions

Slice does not offer a dedicated union construct because it is redundant. By deriving classes from a common base class, you can create the same effect as with a union:

```
interface ShapeShifter {
    Shape translate(Shape s, long xDistance, long yDistance);
};
```

The parameter `s` of the `translate` operation can be viewed as a union of two members: a `Circle` and a `Rectangle`. The receiver of a `Shape` instance can use the type ID (see Section 4.12) of the instance to decide whether it received a `Circle` or a `Rectangle`. Alternatively, if you want something more along the lines of a conventional discriminated union, you can use the following approach:

```
class UnionDiscriminator {
    int d;
};

class Member1 extends UnionDiscriminator {
    // d == 1
    string s;
    float f;
};

class Member2 extends UnionDiscriminator {
    // d == 2
    byte b;
    int i;
};
```

With this approach, the `UnionDiscriminator` class provides a discriminator value. The “members” of the union are the classes that are derived from `UnionDiscriminator`. For each derived class, the discriminator takes on a distinct

value. The receiver of such a union uses the discriminator value in a `switch` statement to select the active union member.

4.9.5 Self-Referential Classes

Classes can be self-referential. For example:

```
class Link {  
    SomeType value;  
    Link next;  
};
```

This looks very similar to the self-referential interface example on page 91, but the semantics are very different. Note that `value` and `next` are data members, not operations, and that the type of `next` is `Link` (*not* `Link*`). As you would expect, this forms the same linked list arrangement as the `Link` interface on page 91: each instance of a `Link` class contains a `next` member that points at the next link in the chain; the final link's `next` member contains a null value. So, what looks like a class including itself really expresses pointer semantics: the `next` data member contains a pointer to the next link in the chain.

You may be wondering at this point what the difference is then between the `Link` interface on page 91 and the `Link` class on page 101. The difference is that classes have *value* semantics, whereas proxies have *reference* semantics. To illustrate this, consider the `Link` *interface* from page 91 once more:

```
interface Link {  
    nonmutating SomeType getValue();  
    nonmutating Link* next();  
};
```

Here, `getValue` and `next` are both operations and the return value of `next` is `Link*`, that is, `next` returns a *proxy*. A proxy has *reference* semantics, that is, it denotes an object somewhere. If you invoke the `getValue` operation on a `Link` proxy, a message is sent to the (possibly remote) servant for that proxy. In other words, for proxies, the object stays put in its server process and we access the state of the object via remote procedure calls. Compare this with the definition of our `Link` *class*:

```
class Link {  
    SomeType value;  
    Link next;  
};
```

Here, `value` and `next` are data members and the type of `next` is `Link`, which has *value* semantics. In particular, while `next` looks and feels like a pointer, *it cannot denote an instance in a different address space*. This means that if we have a chain of `Link` instances, all of the instances are in our local address space and, when we read or write a `value` data member, we are performing local address space operations. This means that an operation, such as `getHead`, that returns a `Link` instance does not just return the head of the chain, *but the entire chain*, as shown in Figure 4.8.

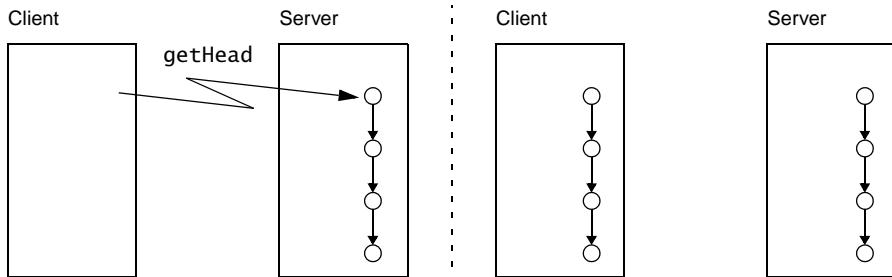


Figure 4.8. Class version of `Link` before and after calling `getHead`.

On the other hand, for the interface version of `Link`, we do not know where all the links are physically implemented. For example, a chain of four links could have each object instance in its own physical server process; those server processes could be each in a different continent. If you have a proxy to the head of this four-link chain and traverse the chain by invoking the `next` operation on each link, you will be sending four remote procedure calls, one to each object.

Self-referential classes are particularly useful to model graphs. For example, we can create a simple expression tree along the following lines:

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {};

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
```

```
    Node operand2;
};

class Operand extends Node {
    long val;
};
```

The expression tree consists of leaf nodes of type `Operand`, and interior nodes of type `UnaryOperator` and `BinaryOperator`, with one or two descendants, respectively. All three of these classes are derived from a common base class `Node`. Note that `Node` is an empty class. This is one of the few cases where an empty base class is justified. (See the discussion on page 92; once we add operations to this class hierarchy (see Section 4.9.7), the base class is no longer empty.)

If we write an operation that, for example, accepts a `Node` parameter, passing that parameter results in transmission of the entire tree to the server:

```
interface Evaluator {
    long eval(Node expression); // Send entire tree for evaluation
};
```

Self-referential classes are not limited to acyclic graphs; the Ice run time permits loops: it ensures that no resources are leaked and that infinite loops are avoided during marshaling.

4.9.6 Classes Versus Structures

One obvious question to ask is: why does Ice provide structures as well as classes, when classes obviously can be used to model structures? The answer has to do with the cost of implementation: classes provide a number of features that are absent for structures:

- Classes support inheritance.
- Classes can be self-referential.
- Classes can have operations (see Section 4.9.7).
- Classes can implement interfaces (see Section 4.9.9).

Obviously, an implementation cost is associated with the additional features of classes, both in terms of the size of the generated code and the amount of memory and CPU cycles consumed at run time. On the other hand, structures are simple collections of values (“plain old structs”) and are implemented using very efficient mechanisms. This means that, if you use structures, you can expect better performance and smaller memory footprint than if you would use classes (especially for

languages with direct support for “plain old structures”, such as C++ and C#). Use a class only if you need at least one of its more powerful features.

4.9.7 Classes with Operations

Classes, in addition to data members, can have operations. The syntax for operation definitions in classes is identical to the syntax for operations in interfaces. For example, we can modify the expression tree from Section 4.9.5 as follows:

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {
    idempotent long eval();
};

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand {
    long val;
};
```

The only change compared to the version in Section 4.9.5 is that the `Node` class now has an `eval` operation. The semantics of this are as for a virtual member function in C++: each derived class inherits the operation from its base class and can choose to override the operation’s definition. For our expression tree, the `Operand` class provides an implementation that simply returns the value of its `val` member, and the `UnaryOperator` and `BinaryOperator` classes provide implementations that compute the value of their respective subtrees. If we call `eval` on the root node of an expression tree, it returns the value of that tree, regardless of whether we have a complex expression or a tree that consists of only a single `Operand` node.

Operations on classes are always executed in the caller’s address space, that is, operations on classes are *local* operations. Therefore, calling an operation on a

class does not ever result in a remote procedure call. Of course, this immediately raises an interesting question: what happens if a client receives a class instance with operations from a server, but client and server are implemented in different languages? Classes with operations require the receiver to supply a factory for instances of the class. The Ice run time only marshals the data members of the class. If a class has operations, the receiver of the class must provide a class factory that can instantiate the class in the receiver's address space, and the receiver is responsible for providing an implementation of the class's operations.

Therefore, if you use classes with operations, it is understood that client and server each have access to an implementation of the class's operations. No code is shipped over wire (which, in an environment of heterogeneous nodes using different operating systems and languages is infeasible).

4.9.8 Architectural Implications of Classes

Classes have a number of architectural implications that are worth exploring in some detail.

Classes without Operations

Classes that do not use inheritance and only have data members (whether self-referential or not) pose no architectural problems: they simply are values that are marshaled like any other value, such as a sequence, structure, or dictionary.

Classes using derivation also pose no problems: if the receiver of a derived instance has knowledge of the derived type, it simply receives the derived type; otherwise, the instance is sliced to the most-derived type that is understood by the receiver. This makes class inheritance useful as a system is extended over time: you can create derived class without having to upgrade all parts of the system at once.

Classes with Operations

Classes with operations require additional thought. Here is an example: suppose that you are creating an Ice application. Also assume that the Slice definitions use quite a few classes with operations. You sell your clients and servers (both written in Java) and end up with thousands of deployed systems.

As time passes and requirements change, you notice a demand for clients written in C++. For commercial reasons, you would like to leave the development of C++ clients to customers or a third party but, at this point, you discover a glitch: your application has lots of classes with operations along the following lines:

```
class ComplexThingForExpertsOnly {  
    // Lots of arcane data members here...  
    MysteriousThing mysteriousOperation(/* parameters */);  
    ArcaneThing arcaneOperation(/* parameters */);  
    ComplexThing complexOperation(/* parameters */);  
    // etc...  
};
```

It does not matter what exactly these operations do. (Presumably, you decided to off-load some of the processing for your application onto the client side for performance reasons.) Now that you would like other developers to write C++ clients, it turns out that your application will work only if these developers provide implementations of all the client-side operations and, moreover, if the semantics of these operations exactly match the semantics of your Java implementations. Depending on what these operations do, providing exact semantic equivalents in a different language may not be trivial, so you decide to supply the C++ implementations yourself. But now, you discover another problem: the C++ clients need to be supported for a variety of operating systems that use a variety of different C++ compilers. Suddenly, your task has become quite daunting: you really need to supply implementations for all the combinations of operating systems and compiler versions that are used by clients. Given the different state of compliance with the ISO C++ standard of the various compilers, and the idiosyncrasies of different operating systems, you may find yourself facing a development task that is much larger than anticipated. And, of course, the same scenario will arise again should you need client implementations in yet another language.

The moral of this story is not that classes with operations should be avoided; they can provide significant performance gains and are not necessarily bad. But, keep in mind that, once you use classes with operations, you are, in effect, using client-side native code and, therefore, you can no longer enjoy the implementation transparencies that are provided by interfaces. This means that classes with operations should be used only if you can tightly control the deployment environment of clients. If not, you are better off using interfaces and classes without operations. That way, all the processing stays on the server and the contract between client and server is provided solely by the Slice definitions, not by the semantics of the additional client-side code that is required for classes with operations.

Classes for Persistence

Ice also provides a built-in persistence mechanism that allows you to store the state of a class in a database with very little implementation effort. To get access to these persistence features, you must define a Slice class whose members store

the state of the class. We discuss the persistence features of Slice in detail in Chapter 22.

4.9.9 Classes Implementing Interfaces

A Slice class can also be used as a servant in a server, that is, an instance of a class can be used to provide the behavior for an interface, for example:

```
interface Time {  
    nonmutating TimeOfDay getTime();  
    idempotent void setTime(TimeOfDay time);  
};  
  
class Clock implements Time {  
    TimeOfDay time;  
};
```

The `implements` keyword indicates that the class `Clock` provides an *implementation* of the `Time` interface. The class can provide data members and operations of its own; in the preceding example, the `Clock` class stores the current time that is accessed via the `Time` interface. A class can implement several interfaces, for example:

```
interface Time {  
    nonmutating TimeOfDay getTime();  
    idempotent void setTime(TimeOfDay time);  
};  
  
interface Radio {  
    idempotent void setFrequency(long hertz);  
    idempotent void setVolume(long dB);  
};  
  
class RadioClock implements Time, Radio {  
    TimeOfDay time;  
    long hertz;  
};
```

The class `RadioClock` implements both `Time` and `Radio` interfaces.

A class, in addition to implementing an interface, can also extend another class:

```

interface Time {
    nonmutating TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};

class Clock implements Time {
    TimeOfDay time;
};

interface AlarmClock extends Time {
    nonmutating TimeOfDay getAlarmTime();
    idempotent void setAlarmTime(TimeOfDay alarmTime);
};

interface Radio {
    idempotent void setFrequency(long hertz);
    idempotent void setVolume(long dB);
};

class RadioAlarmClock extends Clock
    implements AlarmClock, Radio {
    TimeOfDay alarmTime;
    long hertz;
};

```

These definitions result in the inheritance graph shown in Figure 4.9:

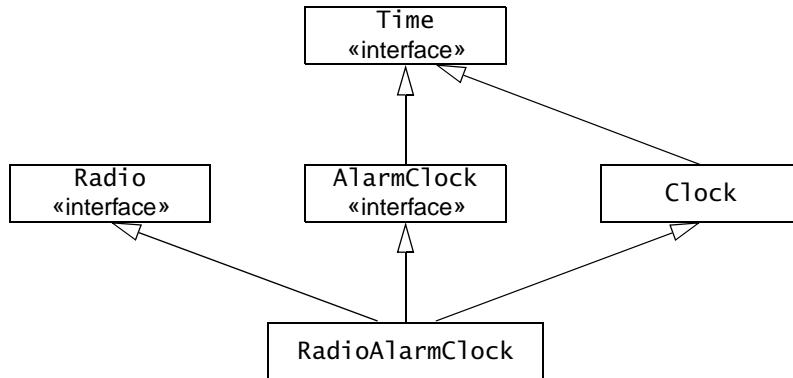


Figure 4.9. A Class using implementation and interface inheritance.

For this definition, `Radio` and `AlarmClock` are abstract interfaces, and `Clock` and `RadioAlarmClock` are concrete classes. As for Java, a class can implement multiple interfaces, but can extend at most one class.

4.9.10 Class Inheritance Limitations

As for interface inheritance, a class cannot redefine an operation or data member that it inherits from a base interface or class. For example:

```
interface BaseInterface {  
    void op();  
};  
  
class BaseClass {  
    int member;  
};  
  
class DerivedClass extends BaseClass implements BaseInterface {  
    void someOperation();      // OK  
    int op();                 // Error!  
    int someMember;           // OK  
    long member;              // Error!  
};
```

4.9.11 Pass-by-Value versus Pass-by-Reference

As we saw in Section 4.9.5, classes naturally support pass-by-value semantics: passing a class transmits the data members of the class to the receiver. Any changes made to these data members by the receiver affect only the receiver's copy of the class; the data members of the sender's class are not affected by the changes made by the receiver.

In addition to passing a class by value, you can pass a class by reference. For example:

```
class TimeOfDay {  
    short hour;  
    short minute;  
    short second;  
    string format();  
};
```

```
interface Example {
    TimeOfDay* get(); // Note: returns a proxy!
};
```

Note that the `get` operation returns a *proxy* to a `TimeOfDay` class and not a `TimeOfDay` instance itself. The semantics of this are as follows:

- When the client receives a `TimeOfDay` proxy from the `get` call, it holds a proxy that differs in no way from an ordinary proxy for an interface.
- The client can invoke operations via the proxy, but *cannot* access the data members. This is because proxies do not have the concept of data members, but represent interfaces: even though the `TimeOfDay` class has data members, only its *operations* can be accessed via a the proxy.

The net effect is that, in the preceding example, the server holds an instance of the `TimeOfDay` class. A proxy for that instance was passed to the client. The only thing the client can do with this proxy is to invoke the `format` operation. The implementation of that operation is provided by the server and, when the client invokes `format`, it sends an RPC message to the server just as it does when it invokes an operation on an interface. The implementation of the `format` operation is entirely up to the server. (Presumably, the server will use the data members of the `TimeOfDay` instance it holds to return a string containing the time to the client.)

The preceding example looks somewhat contrived for classes only. However, it makes perfect sense if classes implement interfaces: parts of your application can exchange class instances (and, therefore, state) by value, whereas other parts of the system can treat these instances as remote interfaces. For example:

```
interface Time {
    string format();
    // ...
};

class TimeOfDay implements Time {
    short hour;
    short minute;
    short second;
};

interface I1 {
    TimeOfDay get();           // Pass by value
    void put(TimeOfDay time); // Pass by value
};
```

```
interface I2 {  
    Time* get(); // Pass by reference  
};
```

In this example, clients dealing with interface I1 are aware of the TimeOfDay class and pass it by value whereas clients dealing with interface I2 deal only with the Time interface. However, the actual implementation of the Time interface in the server uses TimeOfDay instances.

Be careful when designing systems that use such mixed pass-by-value and pass-by-reference semantics. Unless you are clear about what parts of the system deal with the interface (pass by reference) aspects and the class (pass by value) aspects, you can end up with something that is more confusing than helpful.

A good example of putting this feature to use can be found in Freeze (see Chapter 22), which allows you to add classes to an existing interface to implement persistence.

4.10 Forward Declarations

Both interfaces and classes can be forward declared. Forward declarations permit the creation of mutually dependent objects, for example:

```
interface Child; // Forward declaration  
  
sequence<Child*> Children; // OK  
  
interface Parent {  
    Children getChildren(); // OK  
};  
  
interface Child { // Definition  
    Parent* getMother();  
    Parent* getFather();  
};
```

Without the forward declaration of Child, the definition obviously could not compile because Child and Parent are mutually dependent interfaces. You can use forward-declared interfaces and classes to define types (such as the Children sequence in the previous example). Forward-declared interfaces and classes are also legal as the type of a structure, exception, or class member, as the value type of a dictionary, and as the parameter and return type of an operation. However,

you cannot inherit from a forward-declared interface or class until after its definition has been seen by the compiler:

```
interface Base;                      // Forward declaration
interface Derived1 extends Base {};    // Error!
interface Base {};                   // Definition
interface Derived2 extends Base {};    // OK, definition was seen
```

Not inheriting from a forward-declared base interface or class until its definition is seen is necessary because, otherwise, the compiler could not enforce that derived interfaces must not redefine operations that appear in base interfaces.¹⁰

4.11 Modules

A common problem in large systems is pollution of the global namespace: over time, as isolated systems are integrated, name clashes become quite likely. Slice provides the `module` construct to alleviate this problem:

```
module MutableRealms {
    module WishClient {
        // Definitions here...
    };
    module WishServer {
        // Definitions here...
    };
};
```

A module can contain any legal Slice construct, including other module definitions. Using modules to group related definitions together avoids polluting the global namespace and makes accidental name clashes quite unlikely. (You can use a well-known name, such as a company or product name, as the name of the outermost module.)

Modules can be reopened:

¹⁰A multi-pass compiler could be used, but the added complexity is not worth it.

```
module MutableRealms {  
    // Definitions here...  
};  
  
// Possibly in a different source file:  
  
module MutableRealms { // OK, reopened module  
    // More definitions here...  
};
```

Reopened modules are useful for larger projects: they allow you to split the contents of a module over several different source files. The advantage of doing this is that, when a developer makes a change to one part the module, only files dependent on the changed part need be recompiled (instead of having to recompile all files that use the module).

Modules map to C++ namespaces and Java packages, respectively. This allows you to use an appropriate C++ `using` or Java `import` declaration to avoid excessively long identifiers in the source code.

The scope-qualification operator `::` allows you to refer to a type in a non-local scope. For example:

```
module Types {  
    sequence<long> LongSeq;  
};  
  
module MyApp {  
    sequence<Types::LongSeq> NumberTree;  
};
```

Here, the qualified name `Types::LongSeq` refers to `LongSeq` defined in module `Types`. The global scope is denoted by a leading `::`, so we could also refer to `LongSeq` as `::Types::LongSeq`.

The scope-qualification operator also allows you to create mutually dependent interfaces that are defined in different modules. The obvious attempt to do this fails:

```
module Parents {  
    interface Children::Child; // Syntax error!  
    interface Mother {  
        Children::Child* getChild();  
    };  
    interface Father {  
        Children::Child* getChild();  
    };
```

```

};

module Children {
    interface Child {
        Parents::Mother* getMother();
        Parents::Father* getFather();
    };
}

```

This fails because it is syntactically illegal to forward-declare an interface in a different module. To make it work, we must use a reopened module:

```

module Children {
    interface Child;                                // Forward declaration
};

module Parents {
    interface Mother {
        Children::Child* getChild();      // OK
    };
    interface Father {
        Children::Child* getChild();      // OK
    };
};

module Children {                                // Reopen module
    interface Child {                            // Define Child
        Parents::Mother* getMother();
        Parents::Father* getFather();
    };
}

```

While this technique works, it is probably of dubious value: mutually dependent interfaces are, by definition, tightly coupled. On the other hand, modules are meant to be used to place related definitions into the same module, and unrelated definitions into different modules. Of course, this begs the question: if the interfaces are so closely related that they depend on each other, why are they defined in different modules? In the interest of clarity, you probably should avoid this construct, even though it is legal.

4.12 Type IDs

Each Slice type you define has an internal type identifier, known as its *type ID*. The type ID is simply the fully-qualified name of each type. For example, the type ID of the `Child` interface in the preceding example is `::Children::Child`. All type IDs start with a leading `::`, so the type ID of the `Parents` module is `::Parents` (not `Parents`). In general, a type ID is formed by starting with the global scope (`::`) and forming the fully-qualified name of a type by appending each module name in which the type is nested, and ending with the name of the type itself; the components of the type ID are separated by `::`.

Type IDs are used internally by the Ice run time as a unique identifier for each type. For example, when an exception is raised, the marshaled form of the exception that is returned to the client is preceded by its Type ID on the wire. The client-side run time first reads the Type ID and, based on that, unmarshals the remainder of the data as appropriate for the type of the exception.

Type IDs are also used by the `ice_isA` operation (see page 116).

4.13 Operations on Object

The `Object` interface has a number of operations. We cannot define type `Object` in Slice because `Object` is a keyword; regardless, here is what (part of) the definition of `Object` would look like if it were legal:

```
sequence<string> StrSeq;

interface Object {                                     // "Pseudo" Slice!
    void    ice_ping();
    bool   ice_isA(string typeID);
    string ice_id();
    StrSeq ice_ids();
    // ...
};
```

Note that, apart from the illegal use of the keyword `Object` as the interface name, the operation names all contain an underscore. This is deliberate: by putting an underscore into these operation names, it becomes impossible for these built-in operations to ever clash with a user-defined operation. This means that all Slice interfaces can inherit from `Object` without name clashes. There are three built-in operations that are commonly used:

- **ice_ping**

All interfaces support the `ice_ping` operation. That operation is useful for debugging because it provides a basic reachability test for an object: if the object exists and a message can successfully be dispatched to the object, `ice_ping` simply returns without error. If the object cannot be reached or does not exist, `ice_ping` throws a run-time exception that provides the reason for the failure.

- **ice_isA**

The `ice_isA` operation accepts a type identifier (such as the identifier returned by `ice_id`) and tests whether the target object supports the specified type, returning `true` if it does. You can use this operation to check whether a target object supports a particular type. For example, referring to Figure 4.7 once more, assume that you are holding a proxy to a target object of type `AlarmClock`. Table 4.2 illustrates the result of calling `ice_isA` on that proxy with various arguments:

Table 4.2. Calling `ice_isA` on a proxy denoting an object of type `AlarmClock`.

Argument	Result
<code>::Ice::Object</code>	<code>true</code>
<code>::Clock</code>	<code>true</code>
<code>::AlarmClock</code>	<code>true</code>
<code>::Radio</code>	<code>false</code>
<code>::RadioClock</code>	<code>false</code>

As expected, `ice_isA` returns true for `::Clock` and `::AlarmClock` and also returns true for `::Ice::Object` (because *all* interfaces support that type). Obviously, an `AlarmClock` supports neither the `Radio` nor the `RadioClock` interfaces, so `ice_isA` returns false for these types.

- **ice_id**

The `ice_id` operation returns the type ID (see Section 4.12) of the most-derived type of an interface.

- `ice_ids`

The `ice_ids` operation returns a sequence of type IDs that contains all of the type IDs supported by an interface. For example, for the `RadioClock` interface in Figure 4.7, `ice_ids` returns a sequence containing the type IDs `::Ice::Object`, `::Clock`, `::AlarmClock`, `::Radio`, and `::RadioClock`.

4.14 Local Types

In order to access certain features of the Ice run time, you must use APIs that are provided by libraries. However, instead of defining an API that is specific to each implementation language, Ice defines its APIs in Slice using the `local` keyword. The advantage of defining APIs in Slice is that a single definition suffices to define the API for all possible implementation languages. The actual language-specific API is then generated by the Slice compiler for each implementation language. Types that are provided by Ice libraries are defined using the Slice `local` keyword. For example:

```
module Ice {  
    local interface ObjectAdapter {  
        // ...  
    };  
};
```

Any Slice definition (not just interfaces) can have a `local` modifier. If the `local` modifier is present, the Slice compiler does not generate marshaling code for the corresponding type. This means that a local type can *never* be accessed remotely because it cannot be transmitted between client and server. (The Slice compiler prevents use of `local` types in non-`local` contexts.)

In addition, local interfaces and local classes do *not* inherit from `Ice::Object`. Instead, local interfaces and classes have their own, completely

separate inheritance hierarchy. At the root of this hierarchy is the type `Ice::LocalObject`, as shown in Figure 4.10.

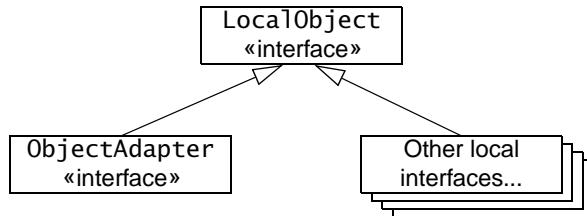


Figure 4.10. Inheritance from `LocalObject`

Because local interfaces form a completely separate inheritance hierarchy, you cannot pass a local interface where a non-local interface is expected and vice-versa.

You rarely need to define local types for your own applications—the `local` keyword exists mainly to allow definition of APIs for the Ice run time. (Because local objects cannot be invoked remotely, there is little point for an application to define local objects; it might as well define ordinary programming-language objects instead.) However, there is one exception to this rule: servant locators must be implemented as local objects (see Section 16.6).

4.15 The Ice Module

APIs for the Ice run time, apart from a small number of language-specific calls that cannot be expressed in Ice, are defined in the `Ice` module. In other words, most of the entire Ice API is actually expressed as Slice definitions. The advantage of doing this is that a single Slice definition is sufficient to define the API for the Ice run time for all supported languages. The respective language mapping rules then determine the exact shape of each Ice API for each implementation language.

We will incrementally explore the contents of the `Ice` module throughout the remainder of this book.

4.16 Names and Scoping

Slice has a number of rules regarding identifiers. You will typically not have to concern yourself with these. However, occasionally, it is good to know how Slice uses naming scopes and looks up identifiers.

4.16.1 Naming Scopes

The following Slice constructs establish a naming scope:

- the global (file) scope
- modules
- interfaces
- classes
- structures
- exceptions
- enumerations
- parameter lists

Within a naming scope, identifiers must be unique, that is, you cannot use the same identifier for different purposes. For example:

```
interface Bad {  
    void op(int p, string p); // Error!  
};
```

Because a parameter list forms a naming scope, it is illegal to use the same identifier `p` for different parameters. Similarly, data members, operation names, interface and class names, etc. must be unique within their enclosing scope.

Within a naming scope, an identifier is introduced at the point of first use; thereafter, within that naming scope, the identifier cannot change meaning. For example:

```
sequence<string> Seq;  
  
interface Bad {  
    Seq op1(); // Seq introduced here  
    int Seq(); // Error, Seq has changed meaning  
};
```

The declaration of `op1` uses `Seq` as its return type, thereby introducing `Seq` into the scope of interface `Bad`. Thereafter, `Seq` can only be used as a type name that denotes a sequence of strings, so the compiler flags the declaration of the second operation as an error.

4.16.2 Case Sensitivity

Identifiers that differ only in case are considered identical, so you must use identifiers that differ not only in capitalization within a naming scope. For example:

```
struct Bad {
    int m;
    string M; // Error!
};
```

The Slice compiler also enforces consistent capitalization for identifiers. Once you have defined an identifier, you must use the same capitalization for that identifier thereafter. For example, the following is in error:

```
sequence<string> StringSeq;

interface Bad {
    stringSeq op(); // Error!
};
```

Note that identifiers must not differ from a Slice keyword in case only. For example, the following is in error:

```
interface Module { // Error, "module" is a keyword
    // ...
};
```

4.16.3 Names in Nested Scopes

Names defined in an enclosing scope can be redefined in an inner scope. For example, the following is legal:

```
module Outer {
    sequence<string> Seq;

    module Inner {
        sequence<short> Seq;
    };
}
```

Within module `Inner`, the name `Seq` refers to a sequence of `short` values and hides the definition of `Outer::Seq`. You can still refer to the other definition by using explicit scope qualification, for example:

```
module Outer {
    sequence<string> Seq;

    module Inner {
        sequence<short> Seq;

        struct Confusing {
            Seq           a;      // Sequence of short
            ::Outer::Seq b;      // Sequence of string
        };
    };
}
```

Needless to say, you should try to avoid such redefinitions—they make it harder for the reader to follow the meaning of a specification.

Same-named constructs cannot be directly nested inside each other. For example, a module named `M` cannot contain any construct also named `M`. The same is true for interfaces, classes, structures, exceptions, and operations. For example, the following examples are all in error:

```
module M {
    interface M { /* ... */ }; // Error!
};

interface I {
    void I();                // Error!
    void op(string op);      // Error!
};

struct S {
    long s;                  // Error, even if case differs!
};
```

The reason for this restriction is that nested types that have the same name are difficult to map into some languages. For example, C++ and Java reserve the name of a class as the name of the constructor, so an interface `I` could not contain an operation named `I` without artificial rules to avoid the name clash. In the interest of simplicity, Slice simply prohibits such constructs.

4.16.4 Name Lookup Rules

When searching for the definition of a name, the compiler first searches backward in the current scope of a definition of the name. If it can find the name in the current scope, it uses that definition. Otherwise, the compiler successively searches enclosing scopes for the name until it reaches the global scope. Here is an example to illustrate this:

```
sequence<double> Seq;

module M1 {
    sequence<string> Seq;           // OK, hides ::Seq

    interface Base {
        Seq op1();                  // Returns sequence of string
    };
}

module M2 {
    interface Derived extends M1::Base {
        Seq op2();                  // Returns sequence of double
    };

    sequence<bool> Seq;            // OK, hides ::Seq

    interface I {
        Seq op();                   // Returns sequence of bool
    };
}

interface I {
    Seq op();                     // Returns sequence of double
};
```

Note that `M2::Derived::op2` returns a sequence of `double`, even though `M1::Base::op1` returns a sequence of `string`. That is, the meaning of a type in a base interface is irrelevant to determining its meaning in a derived interface—the compiler *always* searches for a definition only in the current scope and enclosing scopes, and *never* takes the meaning of a name from a base interface or class.

4.17 Metadata

Slice has the concept of a *metadata* directive. For example:

```
["java:type:java.util.LinkedList"] sequence <int> IntSeq;
```

A metadata directive can appear as a prefix to any Slice definition. Metadata directives appear in a pair of square brackets and contain one or more string literals separated by commas. For example, the following is a syntactically valid metadata directives containing two strings:

```
["a", "b"] interface Example {};
```

Metadata directives are not part of the Slice language per se: the presence of a metadata directive has no effect on the client–server contract, that is, metadata directives do not change the Slice type system in any way. Instead, metadata directives are targeted at specific back-ends, such as the code generator for a particular language mapping. In the preceding example, the `java:` prefix indicates that the directive is targeted at the Java code generator.

Metadata directives permit you to provide supplementary information that does not change the Slice types being defined, but somehow influences how the compiler will generate code for these definitions. For example, a metadata directive `java:type:java.util.LinkedList` instructs the Java code generator to map a sequence to a linked list instead of an array (which is the default).

Metadata directives are also used to create proxies and skeletons that support *Asynchronous Method Invocation (AMI)* and *Asynchronous Method Dispatch (AMD)* (see Chapter 17).

Apart from metadata directives that are attached to a specific definition, there are also global metadata directives. For example:

```
[["java:package:com.acme"]]
```

Note that a global metadata directive is enclosed by double square brackets, whereas a local metadata directive (one that is attached to a specific definition) is enclosed by single square brackets. Global metadata directives are used to pass instructions that affect the entire compilation unit. For example, the preceding metadata directive instructs the Java code generator to generate the contents of the source file into the Java package `com.acme`. Global metadata directives must precede any definitions in a file (but can appear following any `#include` directives).

We discuss specific metadata directives in the relevant chapters to which they apply.

4.18 Using the Slice Compilers

Ice provides a separate Slice compiler for each language mapping. For the C++ mapping, the executable is **slice2cpp**; for the Java mapping, it is **slice2java**. The command-line syntax for either compiler is

<compiler-name> [options] file...

Regardless of which compiler you use, a number of command-line options are common to the compilers for any language mapping. (See the appropriate language mapping chapter for options that are specific to a particular language mapping.) The common command-line options are:

- **-h, --help**

Displays a help message.

- **-v, --version**

Displays the compiler version.

- **-DNAME¹¹**

Defines the preprocessor symbol **NAME**.

- **-DNAME=DEF¹¹**

Defines the preprocessor symbol **NAME** with the value **DEF**.

- **-UNAME¹¹**

Undefines the preprocessor symbol **NAME**.

- **-IDIR**

Add the directory **DIR** to the search path for `#include` directives.

- **--include-dir DIR**

Use **DIR** as the directory to search for Slice definitions provided by Ice.

- **--output-dir DIR**

Place the generated files into directory **DIR**.

- **--dll-export SYMBOL**

Use **SYMBOL** for DLL exports (ignored on non-Windows platforms).

¹¹Note that the preprocessor may not be supported in future versions of Slice, so use these options with caution.

- **-d, --debug**

Print debug information showing the operation of the Slice parser.

- **--ice**

Permit use of the normally reserved prefix `Ice` for identifiers. Use this option only when compiling the source code for the Ice run time.

The Slice compilers permit you to compile more than a single source file, so you can compile several Slice definitions at once, for example:

```
slice2cpp -I. file1.ice file2.ice file3.ice
```

4.19 A Comparison of Slice and CORBA IDL

It is instructive to compare Slice and CORBA IDL because the different feature sets of the two languages illustrate a number of design principles. In this section, we briefly compare the two languages and explain the motivation for the presence or absence of each feature.

Slice is neither a subset nor a superset of CORBA IDL. Instead, Slice both removes and adds features. The overall result is a specification language that is both simpler and more powerful than CORBA IDL, as we will see in the following sections.

4.19.1 Slice Features that are Missing in CORBA IDL

Slice adds a number of features over CORBA IDL. The main additions are:

- Exception inheritance

Lack of inheritance for exceptions in CORBA IDL has long been a thorn in the side of CORBA programmers. The absence of exception inheritance in CORBA IDL prevents natural mappings to languages with equivalent native exception support, such as C++ and Java. In turn, this makes it difficult to implement structured error handling.

As a result, CORBA applications typically use a plethora of exception handlers following each call or block of calls or, at the other extreme, use only generic exception handling at too high a level to still yield useful diagnostics. The trade-off imposed by this is rather draconian: either you have good error handling and diagnostics, but convoluted and difficult-to-maintain code, or you sacrifice error handling in order to keep the code clean.

- Dictionaries

“How do I send a Java hash table to a server?” is one of the most frequently asked questions for CORBA. The standard answer is to model the hash table as a sequence of structures, with each structure containing the key and value, copy the hash table into the sequence, send the sequence, and reconstruct the hash table at the other end.

Doing this is not only wasteful in CPU cycles and memory, but also pollutes the IDL with data types whose presence is motivated by limitations of the CORBA platform (instead of requirements of the application). Slice dictionaries provide support for sending efficient lookup tables as a first-class concept and eliminate the waste and obfuscation of the CORBA approach.

- Nonmutating and idempotent operations

Knowing that an operation will not modify the state of its object permits the Ice run time to transparently recover from transient errors that otherwise would have to be handled by the application. This makes for a more reliable and convenient platform. In addition, nonmutating operations can be mapped to a corresponding construct (if available) in the target language, such as C++ `const` member functions. This improves the static type safety of the system.

- Classes

Slice provides classes that support both pass-by-value and pass-by-proxy semantics. In contrast, CORBA value types (which are somewhat similar) only support pass-by-value semantics: you cannot create a CORBA reference to a value type instance and invoke on that instance remotely.

Slice also uses classes for its automatic persistence mechanism (see Chapter 22). No equivalent feature is provided by CORBA.

- Metadata

Metadata directives permit the language to be extended in a controlled way without affecting the client–server contract. Asynchronous method invocation (AMI) and asynchronous method dispatch (AMD) are examples of the use of metadata directives.

4.19.2 CORBA IDL Features that are Missing in Slice

Slice deliberately drops quite a number of features of CORBA IDL. These can broadly be categorized as follows:

- Redundancies

Some CORBA IDL features are redundant in that they provide more than one way to achieve a single thing. This is undesirable for two reasons:

1. Providing more than one way of achieving the same thing carries a penalty in terms of code and data size. The resulting code bloat also causes performance penalties and so should be avoided.
2. Redundant features are unergonomic and confusing. A single feature is both easier to learn (for programmers) and easier to implement (for vendors) than two features that do the same thing. Moreover, there is always a niggling feeling of unease, especially for newcomers: “How come I can do this in two different ways? When is one style preferable over the other style? Are the two features really identical, or is there some subtle difference I am missing?” Not providing more than one way to do the same thing avoids these questions entirely.

- Non-features

A number of features of CORBA IDL are unnecessary, in the sense that they are hardly ever used. If there is a reasonable way of achieving something without a special-purpose feature, the feature should be absent. This results in a system that easier to learn and use, and a system that is smaller and performs better than it would otherwise.

- Mis-features

Some features of CORBA IDL are mis-features in the sense that they do something that, if not outright wrong, is at least of questionable value. If the potential for abuse through ignorance of a feature is greater than its benefit, the feature should be omitted, again resulting in a simpler, more reliable, and better performing system.

Redundancies

1. Use of the C++ preprocessor

There is no need for a preprocessor for a specification language such as Slice. File inclusion and double-include blocks are the only things that (sensible) people use the preprocessor for when they write CORBA IDL. A simpler and smaller mechanism than the C++ preprocessor can be used instead. Other uses of the preprocessor for CORBA IDL are at best unnecessary or, at worst, obfuscate the code.

2. IDL attributes

IDL attributes are a syntactic short-hand for an accessor operation (for read-only attributes) or a pair of accessor and a modifier operations (for writable attributes). The same thing can be achieved by simply defining an accessor and modifier operation directly.¹²

Attributes introduce considerable complexity into the CORBA run time and APIs. (For example, programmers using the Dynamic Invocation Interface must remember that, to set or get an attribute, they have to use `_get_<attribute-name>` and `_set_<attribute-name>` whereas, for operations, the unadorned name must be used.)

3. Unsigned integers

Unsigned integers add very little to the type system but make it considerably more complex. In addition, if the target programming language does not support unsigned integers (Java does not), it becomes almost impossibly difficult to deal with out-of-range conditions. Currently, Java CORBA programmers deal with the problem by ignoring it. (See BIBREF for a very cogent discussion of the disadvantages of unsigned types.)

4. Underscores in identifiers

Whether or not identifiers should contain underscores is largely a matter of personal taste. However, it is important to have some range of identifiers reserved for the language mapping in order to avoid name clashes. For example, a CORBA IDL specification that contains the identifiers `T` and `T_var` in the same scope cannot be compiled into valid C++. (There are numerous other such conflicts, for C++ as well as other languages.)

Disallowing underscores in Slice identifiers guarantees that all valid Slice specifications can be mapped into valid programming language source code because, at the programming language level, underscores can reliably be used to avoid clashes.

5. Arrays

CORBA IDL offers both arrays and sequences (with sequences further divided into bounded and unbounded sequences). Given that a sequence can easily be used to model an array, there is no need for arrays. Arrays are some-

12.IDL attributes are also second-class citizens because, prior to CORBA 3.x, it was impossible to throw user exceptions from attribute accesses.

what more precise than sequences in that they allow you to state that precisely n elements are required instead of at most n . However, the minute gain in expressiveness is dearly paid for in complexity: not only do arrays contribute to code bloat, they also open a number of holes in the type system. (The CORBA C++ mapping suffers more from the weak type safety of arrays than from any other feature.)

6. Bounded sequences

Much the same arguments that apply to arrays also apply to bounded sequences. The gain in expressiveness of bounded sequences is not worth the complexity that is introduced into language mappings by the feature.

7. Self-referential structures via sequences

CORBA IDL permits structures to have a member that is of the type of its enclosing structure. While the feature is useful, it requires a curiously artificial sequence construct to express. With the introduction of CORBA valuetypes, the feature became redundant because valuetypes support the same thing more clearly and elegantly.

8. Repository ID versioning with `#pragma version`

`#pragma version` directives in CORBA IDL have no purpose whatsoever. The original intent was to allow versioning of interfaces. However, different minor and major version numbers do not actually define any notion of backward (non-)compatibility. Instead, they simply define a new type, which can also be achieved via other means.

Non-Features

1. Arrays

We previously classified arrays as a redundant feature. Experience has shown that arrays are a non-feature as well: after more than ten years of published IDL specifications, you can count the number of places where an array is used on the fingers of one hand.

2. Constant expressions

CORBA IDL allows you to initialize a constant with a constant expression, such as `X * Y`. While this seems attractive, it is unnecessary for a *specification* language. Given that all values involved are compile-time constants, it is entirely possible to work out the values once and write them into the specification directly. (Again, the number of constant expressions in published IDL specifications can be counted on the fingers of one hand.)¹³

3. Types `char` and `wchar`

There simply is no need for a character type in a specification language such as slice. In the rare case where a character is required, type `string` can be used.¹³

4. Fixed-point types

Fixed-point types were introduced into CORBA to support financial applications that need to calculate monetary values. (Floating-point types are ill-suited to this because they cannot store a sufficient number of decimal digits and are subject to a number of undesirable rounding and representational errors.)

The cost of adding fixed-point types to CORBA in terms of code size and API complexity is considerable. Especially for languages without a native fixed-point type, a lot of supporting machinery must be provided to emulate the type. This penalty is paid over and over again, even for languages that are not normally chosen for financial calculations. Moreover, no-one actually does calculations with these types in IDL—instead, IDL simply acts as the vehicle to enable transmission of these types. It is entirely feasible (and simple to implement) to use strings to represent fixed-point values and to convert them into a native fixed-point type in the client and server.

5. Extended floating-point types

While there is a genuine need for extended floating-point types for some applications, the feature is difficult to provide without native support in the underlying hardware. As a result, support for extended floating-point types is widely unimplemented in CORBA. On platforms without extended floating-point support, the type is silently remapped to an ordinary `double`.

Mis-Features

1. `typedef`

IDL `typedef` has probably contributed more to complexity, semantic problems, and bugs in applications than any other IDL feature. The problem with

13. As of version 2.6.x of CORBA, the semantics of IDL constant expressions are still largely undefined: there are no rules for type coercion or how to deal with overflow, and there is no defined binary representation; the result of constant expressions is therefore implementation-dependent.

14. We cannot recall ever having seen a (non-didactic) IDL specification that uses type `char` or `wchar`.

`typedef` is that it does not create a new type. Instead, it creates an alias for an existing type. Judiciously used, type definitions can improve readability of a specification. However, under the hood, the fact that a type can be aliased causes all sorts of problems. For many years, the entire notion of type equivalence was completely undefined in CORBA and it took many pages of specification with complex explanations to nail down the semantic complexities caused by aliased types.¹⁵

The complexity (both in the run time and in the APIs) disappears in absence of the feature: Slice does not permit a type to be aliased, so there can never be any doubt as to the name of a type and, hence, type equivalence.

2. Nested types

IDL allows you define types inside the scope of, for example, an interface or an exception, similar to C++. The amount of complexity created by this decision is staggering: the CORBA specification contains numerous and complex rules for dealing with the order of name lookup, exactly how to decide when a type is introduced into a scope, and how types may (or may not) hide other types of the same name. The complexity carries through into language mappings: nested type definitions result in more complex (and more bulky) source code and are difficult to deal with in languages that do not permit the same nesting of definitions as IDL.¹⁶

Slice allows types to be defined only at global scope and at module scope. Experience shows that this is all that is needed and it avoids all the complexity.

3. Unions

As most OO textbooks will tell you, unions are not required; instead, you can use derivation from a base class to implement the same thing (and enjoy the benefit of type-safe access to the members of the class). IDL unions are yet another source of complexity that is entirely out of proportion to the utility of the feature. Language mappings suffer badly from this. For example, the C++ mapping for IDL unions is something of a challenge even for experts. And, as

15.CORBA 2.6.x still has some unresolved issues with respect to type equivalence.

16.Again, the number of published specifications that actually use nested type definitions can be counted on the fingers of one hand; yet, every CORBA platform must bear the resulting complexity.

with any other feature, unions extract a price in terms of code size and run-time performance.

4. #pragma

IDL allows the use of `#pragma` directives to control the contents of type IDs. This was a most unfortunate choice: because `#pragma` is a preprocessing directive, it is completely outside the normal scoping rules of the language. The resulting complexity is sufficient to take up several pages of explanations in the specification. (And, even with all those explanations, it is still possible to use `#pragma` directives that make no sense, yet cannot be diagnosed as erroneous.)

Slice does not permit control of type IDs because it simply is not necessary.

5. oneway operations

IDL permits an operation to be tagged with a `oneway` keyword. Adding this keyword causes the ORB to dispatch the operation invocation with “best-effort” semantics. In theory, the run time simply fires off the request and then forgets all about it, not caring whether the request is lost or not. In practice, there are a number of problems with this idea:

- `oneway` invocations are delivered via TCP/IP just as normal invocations are. Even though the server may not reply to a `oneway` request, the underlying TCP/IP implementation still attempts to guarantee delivery of the request. This means that, despite the `oneway` keyword, the sending ORB is still subject to flow control (meaning that the client application may block). In addition, at the TCP/IP level, there is return traffic from the server to the client even for a `oneway` request (in the form of acknowledgement and flow-control packets).
- The semantics of `oneway` invocations were poorly defined in earlier versions of CORBA and refined later, to allow the client to have some control over the delivery guarantees. Unfortunately, this resulted in yet more complexity in the application APIs and the ORB implementation.
- Architecturally, the use of `oneway` in IDL is dubious: IDL is an *interface* definition language, but `oneway` has nothing to do with interface. Instead, it controls an aspect of call dispatch that is quite independent of a specific interface. This begs the question of why `oneway` is a first-class language concept when it has nothing to do with the contract between client and server.

Slice does not have a `oneway` keyword, even though it supports `oneway` invocations. This avoids contaminating Slice definitions with non-type related

directives. For asynchronous method invocations, Slice uses metadata directives. The use of such metadata does in no way affect the client–server contract: if you delete all metadata definitions from a specification and then recompile only one of client or server, the interface contract between client and server remains unchanged and valid.

6. IDL contexts

IDL contexts are a general escape hatch that, in essence, permit a sequence of name–string pairs to be sent with every invocation of an operation; the server can examine the name–string pairs and use their contents to change its behavior. Unfortunately, IDL contexts are completely outside the type system and provide no guarantees whatsoever to either client or server: even if an operation has a context clause, there is no guarantee that the client will send any of the named contexts or send well-formed values for those contexts. CORBA has no idea of the meaning or type of these pairs and, therefore, cannot provide any assistance in assuring that their contents are correct (either at compile or run time). The net effect is that IDL contexts shoot a big hole through the IDL type system and result in systems that are hard to understand, code, and debug.

7. Wide strings

CORBA has the notion of support of multiple codesets and character sets for wide strings, with a complex negotiation mechanism that is meant to permit client and server to agree on a particular codeset for transmission of wide strings. A large amount of complexity arises from this choice; as of CORBA 3.0, many ORB implementations still suffer interoperability problems for the exchange of wide string data. The specification still contains a number of unresolved problems that make it unlikely that interoperability will become a reality any time soon.

Slice uses Unicode for its wide strings, that is, there is a single character set and a single defined codeset for the transmission of wide strings. This greatly simplifies the implementation of the run time and avoids interoperability problems.

8. Type Any

The IDL `Any` type is a universal container type that can contain a value of any IDL type. The feature is somewhat similar to exchanging untyped data as a `void *` in C, but improved with introspection, so the type of the contents of an `Any` can be determined at run time. Unfortunately, type `Any` adds much complexity for little gain:

- The API to deal with type Any and its associated type description is arcane and complex. Code that uses type Any is extremely error-prone (particularly in C++). In addition, the language-mapping requires the generation of a large number of helper functions and operators that slow down compilations and take up considerable code and data space at run time.
- Despite the fact that type Any is self-describing and that each instance that is sent over the wire contains a complete (and bulky) description of the value's type, it is impossible for a process to receive and re-transmit a value of type Any without completely unmarshaling and remarshalng the value. This affects programs such as the CORBA Event Service: the extra marshaling cost dominates the overall execution time and limits performance unacceptably for many applications.

Slice uses classes to replace both IDL unions and type Any. This approach is simpler and more type safe than using type Any. In addition, the Slice protocol permits receipt and retransmission of values without forcing the data to be unmarshaled and remarshaled; this leads to smaller and better performing systems than what could be built with CORBA.

9. Anonymous types

Earlier versions of the CORBA specification permitted the use of anonymous IDL types (types that are defined in-line without assigning a separate name to them). Anonymous types caused major problems for language mappings and have since been deprecated in CORBA. However, the complexity of anonymous types is still visible due to backward compatibility concerns. Slice ensures that every type has a name and so prevents any of the problems that are caused by anonymous types.

4.20 Summary

Slice is the fundamental mechanism for defining the client–server contract. By defining data types and interfaces in Slice, you create a language-independent API definition that can be translated by a compiler into an API specific to C++ or Java.

Slice provides the usual built-in types and allows you to create user-defined types of arbitrary complexity, such as sequences, enumerations, structures, dictionaries, and classes. Polymorphism is catered for via inheritance of interfaces, classes, and exceptions. In turn, exceptions provide you with facilities that permit sophisticated error reporting and handling. Modules permit you to group related

parts of a specification and prevent pollution of the global namespace, and metadata can be used to augment Slice definitions with directives for specific compiler backends.

Chapter 5

Slice for a Simple File System

5.1 Chapter Overview

The remainder of this book uses a file system application to illustrate various aspects of Ice. Throughout the book, we progressively improve and modify the application such that it evolves into an application that is realistic and illustrates the architectural and coding aspects of Ice. This allows us to explore the capabilities of the platform to a realistic degree of complexity without overwhelming you with an inordinate amount of detail early on. Section 5.2 outlines the file system functionality, Section 5.3 develops the data types and interfaces that are required for the file system, and Section 5.4 presents the complete Slice definition for the application.

5.2 The File System Application

Our file system application implements a simple hierarchical file system, similar to the file systems we find in Windows or UNIX. To keep code examples to manageable size, we ignore many aspects of a real file system, such as ownership, permissions, symbolic links, and a number of other features. However, we build enough functionality to illustrate how you could implement a fully-featured file system, and we pay attention to things such as performance and scalability. In this

way, we can create an application that presents us with real-world complexity without getting buried in large amounts of code.

To begin with, the file system is non-distributed: although we implement the application in a server that is accessed by clients (so we can access the file system remotely), the initial version requires all files in the file system to be provided by a single server. This means that all directories and files below the root of the file system are implemented in a single server. (We discuss how to remove this limitation and create a truly distributed file system in XREF.)

Our file system consists of directories and files. Directories are containers that can contain either directories or files, meaning that the file system is hierarchical. A dedicated directory is at the root of the file system. Each directory and file has a name. Files and directories with a common parent directory must have different names (but files and directories with different parent directories can have the same name). In other words, directories form a naming scope, and entries with a single directory must have unique names. Directories allow you to list their contents.

For now, we do not have a concept of pathnames, or the creation and destruction of files and directories. Instead, the server provides a fixed number of directories and files. (We will address the creation and destruction of files and directories in XREF.)

Files can be read and written but, for now, reading and writing always replace the entire contents of a file; it is impossible to read or write only parts of a file.

5.3 Slice Definitions for the File System

Given the very simple requirements we just outlined, we can start designing interfaces for the system. Files and directories have something in common: they have a name and both files and directories can be contained in directories. This suggests a design that uses a base type that provides the common functionality, and derived

types that provide the functionality specific to directories and files, as shown in Figure 5.1.

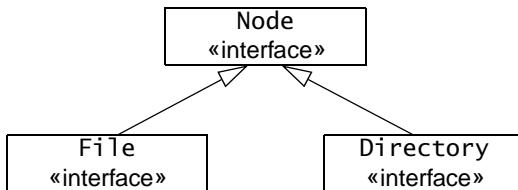


Figure 5.1. Inheritance Diagram of the File System.

The Slice definitions for this look as follows:

```

interface Node {
    // ...
};

interface File extends Node {
    // ...
};

interface Directory extends Node {
    // ...
};
  
```

Next, we need to think about what operations should be provided by each interface. Seeing that directories and files have names, we can add an operation to obtain the name of a directory or file to the `Node` base interface:

```

interface Node {
    nonmutating string name();
};
  
```

The `File` interface provides operations to read and write a file. For the time being, we limit ourselves to text files (see XREF for a discussion of dealing with binary files). For simplicity, we assume that `read` operations never fail and that only `write` operations can encounter error conditions. This leads to the following definitions:

```

exception GenericError {
    string reason;
};

sequence<string> Lines;
  
```

```
interface File extends Node {
    nonmutating Lines read();
    idempotent void write (Lines text) throws GenericError;
};
```

Note that `read` is marked as `nonmutating` because the operation does not modify the state of a file. The `write` operation is marked as `idempotent` because it replaces the entire contents of a file with what is passed in the `text` parameter. This means that it is safe to call the operation with the same parameter value twice in a row: the net result of doing so is the same as having (successfully) called the operation only once.

The `write` operation can raise an exception of type `GenericError`. The exception contains a single `reason` data member, of type `string`. If a `write` operation fails for some reason (such as running out of file system space), the operation throws a `GenericError` exception, with an explanation of the cause of the failure provided in the `reason` data member.

Directories provide an operation to list their contents. Because directories can contain both directories and files, we take advantage of the polymorphism provided by the `Node` base interface:

```
sequence<Node*> NodeSeq;

interface Directory extends Node {
    nonmutating NodeSeq list();
};
```

The `NodeSeq` sequence contains elements of type `Node*`. Because `Node` is a base interface of both `Directory` and `File`, the `NodeSeq` sequence can contain proxies of either type. (Obviously, the receiver of a `NodeSeq` must down-cast each element to either `File` or `Directory` in order to get at the operations provided by the derived interfaces; only the `name` operation in the `Node` base interface can be invoked directly, without doing a down-cast first. Note that, because the elements of `NodeSeq` are of type `Node*` (not `Node`), we are using pass-by-reference semantics: the values returned by the `list` operation are proxies that each point to a remote node on the server.)

These definitions are sufficient to build a simple (but functional) file system. Obviously, there are still some unanswered questions, such as how a client obtains the proxy for the root directory. We will address these questions in XREF.

5.4 The Complete Definition

To avoid polluting the global namespace, we wrap our definitions in a module, resulting in the final definition as follows:

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
        nonmutating Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        nonmutating NodeSeq list();
    };

    interface Filesy {
        nonmutating Directory* getRoot();
    };
}
```

Chapter 6

Client-Side Slice-to-C++ Mapping

6.1 Chapter Overview

In this chapter, we present the client-side Slice-to-C++ mapping (see Chapter 8 for the client-side Slice-to-Java mapping). One part of the client-side C++ mapping concerns itself with rules for representing each Slice data type as a corresponding C++ type; we cover these rules in Section 6.3 to Section 6.10. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Section 6.11 to Section 6.13. Slice classes have the characteristics of both data types and interfaces and are covered in Section 6.14. Finally, we conclude the chapter with a brief comparison of the Slice-to-C++ mapping with the CORBA C++ mapping.

6.2 Introduction

The client-side Slice-to-C++ mapping defines how Slice data types are translated to C++ types, and how clients invoke operations, pass parameters, and handle errors. Much of the C++ mapping is intuitive. For example, Slice sequences map to STL vectors, so there is essentially nothing new you have learn in order to use Slice sequences in C++.

The rules that make up the C++ mapping are simple and regular. In particular, the mapping is free from the potential pitfalls of memory management: all types are self-managed and automatically clean up when instances go out of scope. This means that you cannot accidentally introduce a memory leak by, for example, ignoring the return value of an operation invocation or forgetting to deallocate memory that was allocated by a called operation.

The C++ mapping is fully thread-safe. For example, the reference counting mechanism for classes (see Section 6.14.5) is interlocked against parallel access, so reference counts cannot be corrupted if a class instance is shared among a number of threads. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data—the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Section 6.9 to Section 6.13 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the C++ mapping, you should need no more than the Slice definition of your application and knowledge of the C++ mapping rules. In particular, looking through the generated header files in order to discern how to use the C++ mapping is likely to be confusing because the header files are not necessarily meant for human consumption and, occasionally, contain various cryptic constructs to deal with operating system and compiler idiosyncrasies. Of course, occasionally, you may want to refer to a header file to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

6.3 Mapping for Identifiers

Slice identifiers map to an identical C++ identifier. For example, the Slice identifier `Clock` becomes the C++ identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a C++ keyword, the corresponding C++ identi-

fier is prefixed with `_cpp_`. For example, the Slice identifier `while` is mapped as `_cpp_while`.¹

6.4 Mapping for Modules

Slice modules map to C++ namespaces. The mapping preserves the nesting of the Slice definitions. For example:

```
module M1 {  
    module M2 {  
        // ...  
    };  
    // ...  
};  
  
// ...  
  
module M1 {      // Reopen M1  
    // ...  
};
```

This definition maps to the corresponding C++ definition:

```
namespace M1 {  
    namespace M2 {  
        // ...  
    }  
    // ...  
}  
  
// ...  
  
namespace M1 {  // Reopen M1  
    // ...  
}
```

If a Slice module is reopened, the corresponding C++ namespace is reopened as well.

1. As suggested in Section 4.5.3 on page 60, you should try to avoid such identifiers as much as possible.

6.5 The Ice Namespace

All of the APIs for the Ice run time are nested in the `Ice` namespace, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` namespace are generated from Slice definitions; other parts of the `Ice` namespace provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` namespace throughout the remainder of the book.

6.6 Mapping for Simple Built-In Types

The Slice built-in types are mapped to C++ types as shown in Table 6.1.

Table 6.1. Mapping of Slice built-in types to C++.

Slice	C++
<code>bool</code>	<code>bool</code>
<code>byte</code>	<code>Ice::Byte</code>
<code>short</code>	<code>Ice::Short</code>
<code>int</code>	<code>Ice::Int</code>
<code>long</code>	<code>Ice::Long</code>
<code>float</code>	<code>Ice::Float</code>
<code>double</code>	<code>Ice::Double</code>
<code>string</code>	<code>std::string</code>

Slice `bool` and `string` map to C++ `bool` and `std::string`. The remaining built-in Slice types map to C++ type definitions instead of C++ native types. This allows the Ice run time to provide a definition as appropriate for each target architecture. (For example, `Ice::Int` might be defined as `long` on one architecture and as `int` on another.)

Note that `Ice::Byte` is a `typedef` for `unsigned char`. This guarantees that byte values are always in the range 0..255.

All the basic types are guaranteed to be distinct C++ types, that is, you can safely overload functions that differ in only the types in Table 6.1.

6.7 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

6.7.1 Mapping for Enumerations

Enumerations map to the corresponding enumeration in C++. For example:

```
enum Fruit { Apple, Pear, Orange };
```

Not surprisingly, the generated C++ definition is identical:

```
enum Fruit { Apple, Pear, Orange };
```

6.7.2 Mapping for Structures

Slice structures map to C++ structures with the same name. For each Slice data member, the C++ structure contains a public data member. For example, here is our `Employee` structure from Section 4.7.4 once more:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The Slice-to-C++ compiler generates the following definition for this structure:

```
struct Employee {
    Ice::Long number;
    std::string firstName;
    std::string lastName;
    bool operator==(const Employee&) const;
    bool operator!=(const Employee&) const;
    bool operator<(const Employee&) const;
};
```

For each data member in the Slice definition, the C++ structure contains a corresponding public data member of the same name.

Note that the structure also contains comparison operators. These operators have the following behavior:

- **operator==**

Two structures are equal if (recursively), all its members are equal.

- **operator!=**

Two structures are not equal if (recursively), one or more of its members are not equal.

- **operator<**

The comparison operator treats the members of a structure as sort order criteria: the first member is considered the first criterion, the second member the second criterion, and so on. Assuming that we have two `Employee` structures, `s1` and `s2`, this means that the generated code uses the following algorithm to compare `s1` and `s2`:

```
bool Employee::operator<(const Employee &rhs) const
{
    if (this == &rhs)      // Short-cut self-comparison
        return false;

    // Compare first members
    //
    if (number < rhs.number)
        return true;
    else if (rhs.number < number)
        return false;

    // First members are equal, compare second members
    //
    if (firstName < rhs.firstName)
        return true;
    else if (rhs.firstName < firstName)
        return false;

    // Second members are equal, compare third members
    //
    if (lastName < rhs.lastName)
        return true;
    else if (rhs.lastName < lastName)
        return false;
```

```
// All members are equal, so return false
return false;
}
```

The comparison operators are provided to allow the use of structures as the key type of Slice dictionaries, which are mapped to `std::map` in C++ (see Section 6.7.4).

Note that copy construction and assignment always have deep-copy semantics. You can freely assign structures or structure members to each other without having to worry about memory management. The following code fragment illustrates both comparison and deep-copy semantics:

```
Employee e1, e2;
e1.firstName = "Bjarne";
e1.lastName = "Stroustrup";
e2 = e1;                                // Deep copy
assert(e1 == e2);
e2.firstName = "Andrew";                  // Deep copy
e2.lastName = "Koenig";                   // Deep copy
assert(e2 < e1);
```

Because strings are mapped to `std::string`, there are no memory management issues in this code and structure assignment and copying work as expected. (The default member-wise copy constructor and assignment operator generated by the C++ compiler do the right thing.)

6.7.3 Mapping for Sequences

Here is the definition of our `FruitPlatter` sequence from Section 4.7.3 once more:

```
sequence<Fruit> FruitPlatter;
```

The Slice compiler generates the following C++ definition for the `FruitPlatter` sequence:

```
typedef std::vector<Fruit> FruitPlatter;
```

As you can see, the sequence simply maps to an STL vector. As a result, you can use the sequence like any other STL vector, for example:

```
// Make a small platter with one Apple and one Orange
//
FruitPlatter p;
p.push_back(Apple);
p.push_back(Orange);
```

As you would expect, you can use all the usual STL iterators and algorithms with this vector.

6.7.4 Mapping for Dictionaries

Here is the definition of our `EmployeeMap` from Section 4.7.4 once more:

```
dictionary<long, Employee> EmployeeMap;
```

The following code is generated for this definition:

```
typedef std::map<Ice::Long, Employee> EmployeeMap;
```

Again, there are no surprises here: a Slice dictionary simply maps to an STL map. As a result, you can use the dictionary like any other STL map, for example:

```
EmployeeMap em;
Employee e;

e.number = 42;
e.firstName = "Stan";
e.lastName = "Lipmann";
em[e.number] = e;

e.number = 77;
e.firstName = "Herb";
e.lastName = "Sutter";
em[e.number] = e;
```

Obviously, all the usual STL iterators and algorithms work with this map just as well as with any other STL container.

6.8 Mapping for Constants

Slice constant definitions map to corresponding C++ constant definitions. Here are the constant definitions we saw in Section 4.7.5 on page 69 once more:

```
const bool AppendByDefault = true;
const byte LowerNibble = 0x0f;
const string Advice = "Don't Panic!";
const short TheAnswer = 42;
```

```
const double PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

```
const bool AppendByDefault = true;
const Ice::Byte LowerNibble = 15;
const std::string Advice = "Don't Panic!";
const Ice::Short TheAnswer = 42;
const Ice::Double PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

All constants are initialized directly in the header file, so they are compile-time constants and can be used in contexts where a compile-time constant expression is required, such as to dimension an array or as the `case` label of a `switch` statement.

6.9 Mapping for Exceptions

Here is a fragment of the Slice definition for our world time server from Section 4.8.5 on page 85 once more:

```
exception GenericError {
    string reason;
};

exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exception definitions map as follows:

```
class GenericError: public Ice::UserException {
public:
    std::string reason;

    virtual const std::string & ice_name() const;
    virtual Ice::Exception * ice_clone() const;
    virtual void ice_throw() const;
    // Other member functions here...
};

class BadTimeVal: public GenericError {
```

```

public:
    virtual const std::string & ice_name() const;
    virtual Ice::Exception *     ice_clone() const;
    virtual void                 ice_throw() const;
    // Other member functions here...
};

class BadZoneName: public GenericError {
public:
    virtual const std::string& ice_name() const;
    virtual Ice::Exception *     ice_clone() const;
    virtual void                 ice_throw() const;
};

```

Each Slice exception is mapped to a C++ class with the same name. For each exception member, the corresponding class contains a public data member. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception has three additional member functions:

- `ice_name`

As the name suggests, this member function returns the name of the exception. For example, if you call the `ice_name` member function of a `BadZoneName` exception, it (not surprisingly) returns the string `"BadZoneName"`. The `ice_name` member function is useful if you catch exceptions generically and want to produce a more meaningful diagnostic, for example:

```

try {
    // ...
} catch (const Ice::GenericError &e) {
    cerr << "Caught an exception: " << e.ice_name() << endl;
}

```

If an exception is raised, this code prints the name of the actual exception (`BadTimeVal` or `BadZoneName`) because the exception is being caught by reference (to avoid slicing).

- `ice_clone`

This member function allows you to polymorphically clone an exception. For example:

```
try {
```

```
// ...
} catch (const Ice::UserException & e) {
    Ice::UserException * copy = e.clone();
}
```

`ice_clone` is useful if you need to make a copy of an exception without knowing its precise run-time type. This allows you to remember the exception and throw it later by calling `ice_throw`.

- `ice_throw`

`ice_throw` allows you to throw an exception without knowing its precise run-time type. It is implemented as:

```
void
GenericError::ice_throw() const
{
    throw *this;
}
```

You can call `ice_throw` to throw an exception that you previously cloned with `ice_clone`.

Note that the generated exception classes contain other member functions that are not shown on page 151. However, those member functions are internal to the C++ mapping and are not meant to be called by application code.

All user exceptions ultimately inherit from `Ice::UserException`. In turn, `Ice::UserException` inherits from `Ice::Exception` (which is an alias for `IceUtil::Exception`):

```
namespace IceUtil {
    class Exception {
        virtual const std::string & ice_name() const;
        Exception *               ice_clone() const;
        void                      ice_throw() const;
        virtual void              ice_print(std::ostream &) const;
    };
    std::ostream &operator<<(std::ostream &, const Exception &);
    // ...
}

namespace Ice {
    typedef IceUtil::Exception Exception;

    class UserException: public Exception {
```

```
public:  
    virtual const std::string & ice_name() const = 0;  
    // ...  
};  
}
```

`Ice::Exception` forms the root of the exception inheritance tree. Apart from the usual `ice_name`, `ice_clone`, and `ice_throw` member functions, it contains the `ice_print` member functions. `ice_print` prints the name of the exception. For example, calling `ice_print` on a `BadTimeVal` exception prints:

```
BadTimeVal
```

To make printing more convenient, `operator<<` is overloaded for `Ice::Exception`, so you can also write:

```
try {  
    // ...  
} catch (const Ice::Exception & e) {  
    cerr << e << endl;  
}
```

This produces the same output because `operator<<` calls `ice_print` internally.

For Ice run time exceptions, `ice_print` also shows the file name and line number at which the exception was thrown.

6.10 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice::LocalException` (which, in turn, derives from `Ice::Exception`). `Ice::LocalException` has the usual member functions (`ice_name`, `ice_clone`, `ice_throw`, and (inherited from `Ice::Exception`), `ice_print`, `ice_file`, and `ice_line`).

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 82. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate. For example, a `ConnectTimeoutException` can be handled as any one of the following exception types:

- `Ice::Exception`

This is the root of the complete inheritance tree. Catching `Ice::Exception` catches both user and run-time exceptions.

- `Ice::UserException`

This is the root exception for all user exceptions. Catching `Ice::UserException` catches all user exceptions (but not run-time exceptions).

- `Ice::LocalException`

This is the root exception for all run-time exceptions. Catching `Ice::LocalException` catches all run-time exceptions (but not user exceptions).

- `Ice::TimeoutException`

This is the base exception for both operation-invocation and connection-establishment timeouts.

- `Ice::ConnectTimeoutException`

This exception is raised when the initial attempt to establish a connection to a server times out.

You will probably have little need to catch exceptions by category; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time.

6.11 Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that represents the remote object. This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

6.11.1 Proxy Classes and Proxy Handles

On the client side, interfaces map to classes with member functions that correspond to the operations on those interfaces. Consider the following simple interface:

```
interface Simple {
    void op();
};
```

The Slice compiler generates the following definitions for use by the client:

```
namespace IceProxy {

    class Simple : public virtual IceProxy::Ice::Object {
public:
    void op();
    void op(const Ice::Context &);
    // ...
};

}
```

```
typedef IceInternal::ProxyHandle<IceProxy::Simple> SimplePrx;
```

As you can see, the compiler generates a *proxy class* `Simple` in the `IceProxy` namespace, as well as a *proxy handle* `SimplePrx` in the global namespace. In general, the generated names are `IceProxy::<interface-name>` and `::<interface-name>Prx`. If an interface is nested in a module `M`, the generated names are `IceProxy::M:<interface-name>` and `::M:<interface-name>Prx`.

In the client's address space, an instance of `IceProxy::Simple` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy class instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `Simple` inherits from `IceProxy::Ice::Object`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`. For each operation in the interface, the proxy class has two overloaded member function of the same name. For the preceding example, we find that the operation `op` has been mapped to two member functions `op`.

One of the overloaded member functions has a trailing parameter of type `Ice::Context`. This parameter is for use by the Ice run time to store information about how to deliver a request; normally, you do not need to supply a value here and can pretend that the trailing parameter does not exist. (We examine the `Ice::Context` parameter in detail in Chapter 16. The parameter is also used by IceStorm—see Chapter 27.)

Client-side application code never manipulates proxy class instances directly. In fact, you are not allowed to instantiate a proxy class directly. The following code will not compile because `IceProxy::Simple` contains pure virtual member functions:

```
IceProxy::Simple s;      // Compile-time error!
```

Proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. When the client receives a proxy from the run time, it is given a *proxy handle* to the proxy, of type `<interface-name>Prx` (`SimplePrx` for the preceding example). The client accesses the proxy via its proxy handle; the handle takes care of forwarding operation invocations to its underlying proxy, as well as reference-counting the proxy. This means that no memory-management issues can arise: deallocation of a proxy is automatic and happens once the last handle to the proxy disappears (goes out of scope).

Because the application code always uses proxy handles and never touches the proxy class directly, we usually use the term *proxy* to denote both proxy handle and proxy class. This reflects the fact that, in actual use, the proxy handle looks and feels like the underlying proxy class instance. If the distinction is important, we use the terms *proxy class*, *proxy class instance*, and *proxy handle*.

6.11.2 Methods on Proxy Handles

As we saw for the preceding example, the handle is actually a template of type `IceInternal::ProxyHandle` that takes the proxy class as the template parameter. This template has the usual constructor, copy constructor, and assignment operator:

- Default constructor

You can default-construct a proxy handle. The default constructor creates a proxy that points nowhere (that is, points at no object at all.) If you invoke an operation on such a null proxy, you get an

`IceUtil::NullHandleException`:

```
try {
    SimplePrx s;          // Default-constructed proxy
    s->op();              // Call via nil proxy
    assert(0);              // Can't get here
} catch (const IceUtil::NullHandleException &) {
    cout << "As expected, got a NullHandleException" << endl;
}
```

- Copy constructor

The copy constructor ensures that you can construct a proxy handle from another proxy handle. Internally, this increments a reference count on the proxy; the destructor decrements the reference count again and, once the count drops to zero, deallocates the underlying proxy class instance. That way, memory leaks are avoided:

```
{
    SimplePrx s1 = ...;           // Enter new scope
    SimplePrx s2(s1);            // Get a proxy from somewhere
    assert(s1 == s2);            // Copy-construct s2
                                // Assertion passes
}
                                // Leave scope; s1, s2, and the
                                // underlying proxy instance
                                // are deallocated
```

Note the assertion in this example: proxy handles support comparison (see Section 6.11.3).

- Assignment operator

You can freely assign proxy handles to each other. The handle implementation ensures that the appropriate memory-management activities take place.

Self-assignment is safe and you do not have to guard against it:

```
SimplePrx s1 = ...;           // Get a proxy from somewhere
SimplePrx s2;                 // s2 is nil
s2 = s1;                      // both point at the same object
s1 = 0;                        // s1 is nil
s2 = 0;                        // s2 is nil
```

Widening assignments work implicitly. For example, if we have two interfaces, `Base` and `Derived`, we can widen a `DerivedPrx` to a `BasePrx` implicitly:

```
BasePrx base;
DerivedPrx derived;
base = derived;                // Fine, no problem
derived = base;                // Compile-time error
```

Implicit narrowing conversions result in a compile error, so the usual C++ semantics are preserved: you can always assign a derived type to a base type, but not vice versa.

- Checked cast

Proxy handles provide a `checkedCast` method:

```
namespace IceInternal {
```

```
template<typename T>
class ProxyHandle : public IceUtil::HandleBase<T> {
public:
    template<class Y>
    static ProxyHandle checkedCast(const ProxyHandle<Y> & r);
    // ...
};
```

A checked cast has the same function for proxies as a C++ `dynamic_cast` has for pointers: it allows you to assign a derived proxy to a base proxy. If the base proxy's run-time type is compatible with the derived proxy's static type, the assignment succeeds and, after the assignment, the base proxy denotes the same object as the derived proxy. Otherwise, if the base proxy's run-time type is incompatible with the derived proxy's static type, the derived proxy is set to null. Here is an example to illustrate this:

```
BasePrx base = ...;      // Initialize base proxy
DerivedPrx derived;
derived = DerivedPrx::checkedCast(base);
if (derived) {
    // Base has run-time type Derived,
    // use derived...
} else {
    // Base has some other, unrelated type
}
```

The expression `DerivedPrx::checkedCast(base)` tests whether `base` points at an object of type `Derived`. If so, the cast succeeds and `derived` is set to point at the same object as `base`. Otherwise, the cast fails and `derived` is set to the null proxy.

Note that `checkedCast` is a static member function so, to do a down-cast, you always use the syntax `<interface-name>Prx::checkedCast`.

Also note that you can use proxies in boolean contexts. For example,
`if (proxy)` returns true if the proxy is not null (see Section 6.11.3).

A `checkedCast` typically results in a remote message to the server.² The message effectively asks the server “is the object denoted by this reference of

2. In some cases, the Ice run time can optimize the cast and avoid sending a message. However, the optimization applies only in narrowly-defined circumstances, so you cannot rely on a `checkedCast` not sending a message.

type Derived?" The reply from the server is communicated to the application code in form of a successful (non-null) or unsuccessful (null) result. Sending a remote message is necessary because, as a rule, there is no way for the client to find out what the actual run-time type of a proxy is without confirmation from the server. (For example, the server may replace the implementation of the object for an existing proxy with a more derived one.) This means that you have to be prepared for a `checkedCast` to fail. For example, if the server is not running, you will receive a `ConnectFailedException`; if the server is running, but the object denoted by the proxy no longer exists, you will receive an `ObjectNotExistException`.

- Unchecked cast

In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast:

```
namespace IceInternal {
    template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T> {
        public:
            template<class Y>
            static ProxyHandle uncheckedCast(const ProxyHandle<Y> & r);
            // ...
    };
}
```

An `uncheckedCast` provides a down-cast *without* consulting the server as to the actual run-time type of the object, for example:

```
BasePrx base = ...;      // Initialize to point at a Derived
DerivedPrx derived;
derived = DerivedPrx::uncheckedCast(base);
// Use derived...
```

You should use an `uncheckedCast` only if you are certain that the proxy indeed supports the more derived type: an `uncheckedCast`, as the name implies, is not checked in any way; it does not contact the object in the server and, if it fails, it does not return null. (An unchecked cast is implemented internally like a C++ `static_cast`, no checks of any kind are made). If you use the proxy resulting from an incorrect `uncheckedCast` to invoke an operation, the behavior is undefined. Most likely, you will receive an `ObjectNotExistException` or `OperationNotExistException`, but, depending on the circumstances, the Ice run time may also report an

exception indicating that unmarshaling has failed, or even silently return garbage results.

Despite its dangers, `uncheckedCast` is still useful because it avoids the cost of sending a message to the server. And, particularly during initialization (see Chapter 7), it is common to receive a proxy of static type `Ice::Object`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

6.11.3 Object Identity and Proxy Comparison

Apart from the methods discussed in Section 6.11.2, proxy handles also support comparison. Specifically, the following operators are supported:

- `==`
- `!=`

These operators permit you to compare proxies for equality and inequality. To test whether a proxy is null, use a comparison with the literal 0, for example:

```
if (proxy == 0)
    // It's a nil proxy
else
    // It's a non-nil proxy
```

- `<`

Proxies support operator `<`. This allows you to place proxies into STL containers such as maps or sorted lists.

- Boolean comparison

Proxies have a conversion operator to `bool`. The operator returns true if a proxy is not null, and false otherwise. This allows you to write:

```
BasePrx base = ...;
if (base)
    // It's a non-nil proxy
else
    // It's a nil proxy
```

Note that proxy comparison with the overloaded operators `==`, `!=`, and `<` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `==` and `!=` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

```

Ice::ObjectPrx p1 = ...;           // Get a proxy...
Ice::ObjectPrx p2 = ...;           // Get another proxy...

if (p1 != p2) {
    // p1 and p2 denote different objects          // WRONG!
} else {
    // p1 and p2 denote the same object          // Correct
}

```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `==`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `==`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you must use a helper function in the `Ice` namespace:

```

namespace Ice {

    bool proxyIdentityLess(const ObjectPrx &,
                           const ObjectPrx &);
    bool proxyIdentityEqual(const ObjectPrx &,
                           const ObjectPrx &);

}

```

The `proxyIdentityEqual` function returns true if the object identities embedded in two proxies are the same and ignores other information in the proxies, such as facet and transport information. The `proxyIdentityLess` function establishes a total ordering on proxies. It is provided mainly so you can use object identity comparison with STL sorted containers.

`proxyIdentityEqual` allows you to correctly compare proxies for object identity:

```

Ice::ObjectPrx p1 = ...;           // Get a proxy...
Ice::ObjectPrx p2 = ...;           // Get another proxy...

if (!Ice::proxyIdentityEqual(p1, p2) {

```

```

        // p1 and p2 denote different objects      // Correct
    } else {
        // p1 and p2 denote the same object      // Correct
    }
}

```

6.12 Mapping for Operations

As we saw in Section 6.11, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy handle. For example, here is part of the definitions for our file system from Section 5.4:

```

module Filesystem {
    interface Node {
        nonmutating string name();
    };
    // ...
};

```

The proxy class for the `Node` interface, tidied up to remove irrelevant detail, is as follows:

```

namespace IceProxy {
    namespace Filesystem {
        class Node : virtual public IceProxy::Ice::Object {
            public:
                std::string name();
                // ...
            };
            typedef IceInternal::ProxyHandle<Node> NodePrx;
            // ...
        }
        // ...
    }
}

```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```

NodePrx node = ...;           // Initialize proxy
string name = node->name();   // Get name via RPC

```

The proxy handle overloads `operator->` to forward method calls to the underlying proxy class instance which, in turn, sends the operation invocation to the

server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.

Because the return value is of type `string`, it is safe to ignore the return value. For example, the following code contains no memory leak:

```
NodePrx node = ...;           // Initialize proxy
node->name();                // Useless, but no leak
```

This is true for all mapped Slice types: you can safely ignore the return value of an operation, no matter what its type—return values are always returned by value. If you ignore the return value, no memory leak occurs because the destructor of the returned value takes care of deallocating memory as needed.

6.12.1 Normal, idempotent, and nonmutating Operations

You can add an idempotent or nonmutating qualifier to a Slice operation. As far as the signature for the corresponding proxy methods is concerned, neither `idempotent` nor `nonmutating` have any effect. For example, consider the following interface:

```
interface Example {
    string op1();
    idempotent string op2();
    nonmutating string op3();
};
```

The proxy class for this interface looks like this:

```
namespace IceProxy {
    class Example : virtual public IceProxy::Ice::Object {
public:
    std::string op1();
    std::string op2();      // idempotent
    std::string op3();      // nonmutating
    // ...
};}
```

Because `idempotent` and `nonmutating` affect an aspect of call dispatch, not interface, it makes sense for the three methods to look the same.

6.12.2 Passing Parameters

In-Parameters

The parameter passing rules for the C++ mapping are very simple: parameters are passed either by value (for small values) or by `const` reference (for values that are larger than a machine word). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats—see page 167).

Here is an interface with operations that pass parameters of various types from client to server:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
struct NumberAndString {
    Ice::Int x;
    std::string str;
    // ...
};

typedef std::vector<std::string> StringSeq;

typedef std::map<Ice::Long, StringSeq> StringTable;

namespace IceProxy {
    class ClientToServer : virtual public IceProxy::Ice::Object {
public:
    void op1(Ice::Int, Ice::Float, bool, const std::string &);
    void op2(const NumberAndString &,
            const StringSeq &,
            const StringTable &);
```

```

        void op3(const ClientToServerPrx &);

        // ...
    };
}

```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```

ClientToServerPrx p = ...;                                // Get proxy...

p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14;
bool b = true;
string s = "Hello world!";
p->op1(i, f, b, s);                                     // Pass simple variables

NumberAndString ns = { 42, "The Answer" };
StringSeq ss;
ss.push_back("Hello world!");
StringTable st;
st[0] = ss;
p->op2(ns, ss, st);                                     // Pass complex variables

p->op3(p);                                              // Pass proxy

```

You can pass either literals or variables to the various operations. Because everything is passed by value or `const` reference, there are no memory-management issues to consider.

out-Parameters

The C++ mapping passes `out`-parameters by reference. Here is the Slice definition from page 165 once more, modified to pass all parameters in the `out` direction:

```

struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
}

```

```
    void op2(out NumberAndString ns,
            out StringSeq ss,
            out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
namespace IceProxy {
    class ServerToClient : virtual public IceProxy::Ice::Object {
public:
    void op1(Ice::Int &, Ice::Float &, bool &, std::string &);
    void op2(NumberAndString &, StringSeq &, StringTable &);
    void op3(ServerToClientPrx &);
    // ...
};
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

```
ServerToClientPrx p = ...;           // Get proxy...

int i;
float f;
bool b;
string s;

p->op1(i, f, b, s);
// i, f, b, and s contain updated values now

NumberAndString ns;
StringSeq ss;
StringTable st;

p->op2(ns, ss, st);
// ns, ss, and st contain updated values now

p->op3(p);
// p has changed now!
```

Again, there are no surprises in this code: the caller simply passes variables to an operation; once the operation completes, the values of those variables will be set by the server.

It is worth having another look at the final call:

```
p->op3(p);           // Weird, but well-defined
```

Here, `p` is the proxy that is used to dispatch the call. That same variable `p` is also passed as an `out`-parameter to the call, meaning that the server will set its value. In general, passing the same parameter as both an input and output parameter is safe: the Ice run time will correctly handle all locking and memory-management activities.

Another, somewhat pathological example is the following:

```
sequence<int> Row;
sequence<Row> Matrix;

interface MatrixArithmetic {
    void multiply(Matrix m1,
                  Matrix m2,
                  out Matrix result);
};
```

Given a proxy to a `MatrixArithmetic` interface, the client code could do the following:

```
MatrixArithmeticPrx ma = ...;           // Get proxy...
Matrix m1 = ...;                      // Initialize one matrix
Matrix m2 = ...;                      // Initialize second matrix
ma->squareAndCubeRoot(m1, m2, m1); // !!!
```

This code is technically legal, in the sense that no memory corruption or locking issues will arise, but it has surprising behavior: because the same variable `m1` is passed as an input parameter as well as an output parameter, the final value of `m1` is indeterminate—in particular, if client and server are collocated in the same address space, the implementation of the operation will overwrite parts of the input matrix `m1` in the process of computing the result because the result is written to the same physical memory location as one of the inputs. In general, you should take care when passing the same variable as both an input and output parameter and only do so if the called operation guarantees to be well-behaved in this case.

Chained Invocations

Consider the following simple interface containing two operations, one to set a value and one to get it:

```
interface Name {
    string getName();
    void setName(string name);
};
```

Suppose we have two proxies to interfaces of type `Name`, `p1` and `p2`, and chain invocations as follows:

```
p2->setName(p1->getName());
```

This works exactly as intended: the value returned by `p1` is transferred to `p2`.³ There are no memory-management or exception safety issues with this code.³

6.13 Exception Handling

Any operation invocation may throw a run-time exception (see Section 6.10 on page 154) and, if the operation has an exception specification, may also throw user exceptions (see Section 6.9 on page 151). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as C++ exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

```
ChildPrx p = ...;           // Get proxy...
try {
    p->askToCleanUp();     // Give it a try...
} catch (const Tantrum & t) {
    cout << "The child says: " << t.reason << endl;
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be dealt with by exception handlers higher in the hierarchy. For example:

3. This is worth pointing out because, in CORBA, the equivalent code leaks memory (as does ignoring the return value in many cases).

```

void run()
{
    ChildPrx p = ...; // Get proxy...
    try {
        p->askToCleanUp(); // Give it a try...
    } catch (const Tantrum & t) {
        cout << "The child says: " << t.reason << endl;

        p->scold(); // Recover from error...
    }
    p->praise(); // Give positive feedback...
}

int
main(int argc, char * argv[])
{
    try {
        // ...
        run();
        // ...
    } catch (const Ice::Exception & e) {
        cerr << "Unexpected run-time error: " << e << endl;
        return 1;
    }
    return 0;
}

```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 3.)

For efficiency reasons, you should always catch exceptions by `const` reference. This permits the compiler to avoid calling the exception's copy constructor (and, of course, prevents the exception from being sliced to a base type).

Exceptions and out-Parameters

The Ice run time makes no guarantees about the state of `out`-parameters when an operation throws an exception: the parameter may have still have its original value or may have been changed by the operation's implementation in the target object. In other words, for `out`-parameters, Ice provides the weak exception guarantee [19] but does not provide the strong exception guarantee.⁴

Exceptions and Return Values

For return values, C++ provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown. (Of course, this guarantee holds only if you do not use the same variable as both an `out`-parameter and to receive the return value of an invocation (see page 167).)

6.14 Mapping for Classes

Slice classes are mapped to C++ classes with the same name. The generated class contains a public data member for each Slice data member, and a virtual member function for each operation. Consider the following class definition:

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
    string format();      // Return time as hh:mm:ss
};
```

The Slice compiler generates the following code for this definition:

```
class TimeOfDay : virtual public Ice::Object {
public:
    Ice::Short hour;
    Ice::Short minute;
    Ice::Short second;

    virtual std::string format() = 0;

    virtual bool ice_isA(const std::string &);

    virtual const std::string & ice_id();
    static const std::string & ice_staticId();

    static const Ice::ObjectFactoryPtr & ice_factory();
    // ...
};

typedef IceInternal::Handle<TimeOfDay> TimeOfDayPtr;
```

4. This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice::Object`. This means that all classes implicitly inherit from `Ice::Object`, which is the ultimate ancestor of all classes. Note that `Ice::Object` is *not* the same as `IceProxy::Ice::Object`. In other words, you *cannot* pass a class where a proxy is expected and vice versa. (However, you *can* pass a proxy for the class—see Section 6.14.5.)
2. The generated class contains a public member for each Slice data member.
3. The generated class contains a pure virtual member function for each Slice operation.
4. The generated class contains additional member functions: `ice_isA`, `ice_id`, `ice_staticId`, and `ice_factory`.
5. The compiler generates a type definition `TimeOfDayPtr`. This type implements a smart pointer that wraps dynamically-allocated instances of the class. In general, the name of this type is `<class-name>Ptr`. Do not confuse this with `<class-name>Prx`—that type exists as well, but is the proxy handle for the class, not a smart pointer.

There is quite a bit to discuss here, so we will look at each item in turn.

6.14.1 Inheritance from `Ice::Object`

Like interfaces, classes implicitly inherit from a common base class, `Ice::Object`. However, as shown in Section 6.14.1, classes inherit from `Ice::Object` instead of `Ice::ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

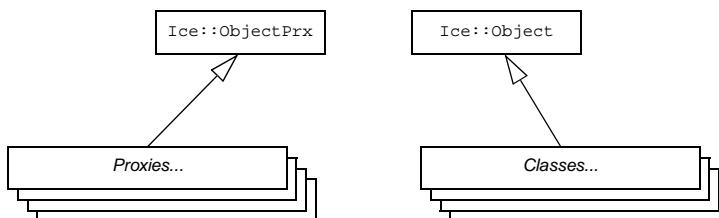


Figure 6.1. Inheritance from `Ice::ObjectPrx` and `Ice::Object`.

`Ice::Object` contains a number of member functions:

```
namespace Ice {
    class Object : virtual public IceUtil::Shared {
public:
    virtual bool ice_isA(const std::string &,
                         const Current & = Current());
    virtual void ice_ping(const Current & = Current());
    virtual const std::string & ice_id(
        const Current & = Current());
    static const std::string & ice_staticId();
    virtual Ice::Int ice_hash() const;

    virtual bool operator==(const Object &) const;
    virtual bool operator!=(const Object &) const;
    virtual bool operator<(const Object &) const;
};
}
```

The member functions of `Ice::Object` behave as follows:

- `operator==`
- `operator!=`
- `operator<`

The comparison operators permit you to use classes as elements of STL sorted containers.

- `ice_hash`

This method returns a hash value for the class, allowing you to easily place classes into hash tables.

- `ice_ping`

As for interfaces, `ice_ping` provides a basic reachability test for the class.

- `ice_id`

This function returns the actual run-time type ID for a class. If you call `ice_id` through a smart pointer to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.

- `ice_staticId`

This function returns the static type ID of a class.

6.14.2 Data Members of Classes

Data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member.

6.14.3 Operations of Classes

Operations on classes are mapped to pure virtual member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

```
class TimeOfDayI : virtual public TimeOfDay {
public:
    virtual std::string format() {
        std::ostringstream s;
        s << setw(2) << setfill('0') << hour;
        s << setw(2) << setfill('0') << minute;
        s << setw(2) << setfill('0') << second;
        return s.c_str();
    }
};
```

6.14.4 Class Factories

Having created a class such as this, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an `out`-parameter from an operation invocation. To see why, consider the following simple interface:

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        void removeObjectFactory(string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
}
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
}
```

The object factory's `create` operation is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` operation is called by the Ice run time when the factory is unregistered or its `Communicator` is destroyed. A possible implementation of our object factory is:

```
class ObjectFactory : public Ice::ObjectFactory {
public:
    virtual Ice::ObjectPtr create(const std::string &) {
        assert(type == "::TimeOfDay");
        return new TimeOfDayI;
    }
    virtual void destroy() {}
};
```

The `create` method is passed the type ID (see Section 4.12) of the class to instantiate. For our `TimeOfDay` class, the type ID is "`::TimeOfDay`". Our implementation of `create` checks the type ID: if it is "`::TimeOfDay`", it instantiates and returns a `TimeOfDayI` object. For other type IDs, it asserts because it does not know how to instantiate other types of objects.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
Ice::CommunicatorPtr ic = ...;
ic->addObjectFactory(new ObjectFactory, "::TimeOfDay");
```

Now, whenever the Ice run time needs to instantiate a class with the type ID "`::TimeOfDay`", it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when you call `Communicator::removeObjectFactory` or when the `Communicator` is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the `Communicator`—if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`. Similarly, attempting to remove the factory for a type ID that is not registered throws a `NotRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

6.14.5 Smart Pointers for Classes

A recurring theme for C++ programmers is the need to deal with memory allocations and deallocations in their programs. The difficulty of doing so is well known: in the face of exceptions, multiple return paths from functions, and callee-allocated memory that must be deallocated by the caller, it can be extremely difficult to ensure that a program does not leak resources. This is particularly important in multi-threaded programs: if you do not rigorously track ownership of dynamic memory, a thread may delete memory that is still used by another thread, usually with disastrous consequences.

To alleviate this problem, Ice provides smart pointers for classes. These smart pointers use reference counting to keep track of each class instance and, when the last reference to a class instance disappears, automatically delete the instance. Smart pointers are generated by the Slice compiler for each class type. For a Slice class `<class-name>`, the compiler generates a C++ smart pointer called `<class-name>Ptr`. Rather than showing all the details of the generated class, here is the basic usage pattern: whenever you allocate a class instance on the heap,

you simply assign the pointer returned from `new` to a smart pointer for the class. Thereafter, memory management is automatic and the class instance is deleted once the last smart pointer for it goes out of scope:

```
{                                     // Open scope
    TimeOfDayPtr tod = new TimeOfDayI;   // Allocate instance
    tod->second = 0;                   // Initialize
    tod->minute = 0;
    tod->hour = 0;
    // Use instance...
}

}                                     // No memory leak here!
```

As you can see, you use `operator->` to access the members of the class via its smart pointer. When the `tod` smart pointer goes out of scope, its destructor runs and, in turn, the destructor takes care of calling `delete` on the underlying class instance, so no memory is leaked.

The smart pointers perform reference counting of their underlying class instance:

- The constructor of a class sets its reference count to zero.
- Initializing a smart pointer with a dynamically-allocated class instance causes the smart pointer to increment the reference count for the class by one.
- Copy constructing a smart pointer increments the reference count for the class by one.
- Assigning one smart pointer to another increments the target's reference count and decrements the source's reference count. (Self-assignment is safe.)
- The destructor of a smart pointer decrements the reference count by one and calls `delete` on its class instance if the reference count drops to zero.

Figure 6.2 shows the situation after default-constructing a smart pointer as follows:

```
TimeOfDayPtr tod;
```

This creates a smart pointer with an internal null pointer.



Figure 6.2. Newly initialized smart pointer.

Constructing a class instance creates that instance with a reference count of zero; the assignment to the class pointer causes the smart pointer to increment the class's reference count:

```
tod = new TimeOfDayI; // Refcount == 1
```

The resulting situation is shown in Figure 6.3.

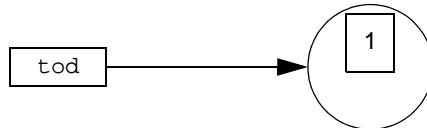


Figure 6.3. Initialized smart pointer.

Assigning or copy-constructor a smart pointer assigns and copy-constructs the smart pointer (not the underlying instance) and increments the reference count of the instance:

```
TimeOfDayPtr tod2(tod); // Copy-construct tod2
TimeOfDayPtr tod3;
tod3 = tod; // Assign to tod3
```

The situation after executing these statements is shown in Figure 6.4:

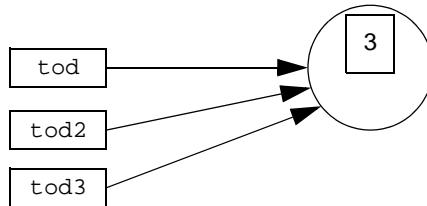


Figure 6.4. Three smart pointers pointing at the same class instance.

Continuing the example, we can construct a second class instance and assign it to one of the original smart pointers, `tod2`:

```
tod2 = new TimeOfDayI;
```

This decrements the reference count of the class originally denoted by `tod2` and increments the reference count of the class that is assigned to `tod2`. The resulting situation is shown in Figure 6.5.

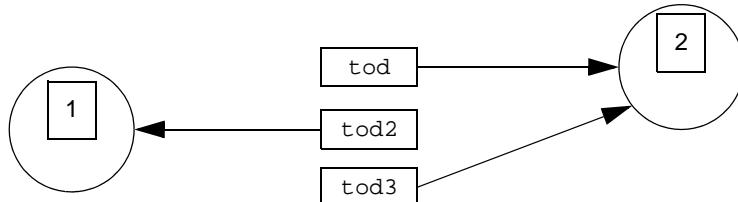


Figure 6.5. Three smart pointers and two instances.

You can clear a smart pointer by assigning zero to it:

```
tod = 0;           // Clear handle
```

As you would expect, this decrements the reference count of the instance, as shown in Figure 6.6.

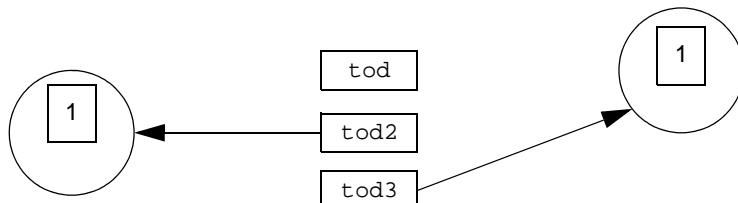


Figure 6.6. Decrementated reference count after clearing a smart pointer.

If a smart pointer goes out of scope, is cleared, or has a new instance assigned to it, the smart pointer decrements the reference count of its instance; if the reference count drops to zero, the smart pointer calls `delete` to deallocate the instance. The following code snippet deallocates the instance on the right by assigning `tod2` to `tod3`:

```
tod3 = tod2;
```

This results in the situation shown in Figure 6.7.

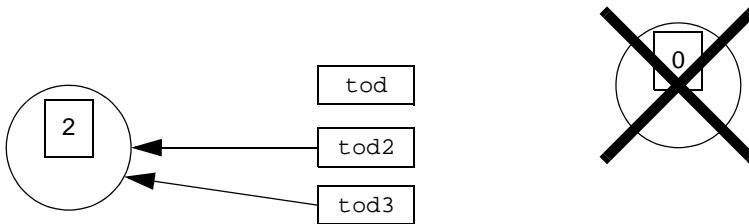


Figure 6.7. Deallocation of an instance with a reference count of zero.

Deep-Copying of Classes Via Smart Pointers

Because assignment of smart pointers only affects the smart pointer, not the underlying instance, you cannot achieve a deep copy of class instances by assigning smart pointers. To make a deep copy, you must implement and invoke a member function on the class, such as a `clone` method. That `clone` method then can return a deep copy of the class instance. For example, to create a deep copy of a `TimeOfDayI` instance, you can write the following:

```

class TimeOfDayI;

typedef IceUtil::Handle<TimeOfDayI> TimeOfDayIPtr;

class TimeOfDayI : virtual public TimeOfDay {
public:
    virtual string format() { /* as before... */ }

    virtual TimeOfDayIPtr clone() {
        TimeOfDayIPtr clone = new TimeOfDayI;
        clone->hour = this->hour;
        clone->minute = this->minute;
        clone->second = this->second;
        return clone;
    }
};
  
```

Note that the code defines `TimeOfDayIPtr` as a `typedef` for `IceUtil::Handle<TimeOfDayI>`. This class is a template that contains the smart pointer implementation. If you want smart pointers for classes that are not generated by the Slice compiler, you must define a smart pointer type as illustrated by this type definition.

The `clone` member function simply makes a copy of its data members and returns a new instance as a smart pointer. Note that you *must* define a `clone` function to make deep copies of classes. This is because all Slice-generated classes disable the assignment operator, to avoid problems when manipulating derived class instances via base class smart pointers: attempts to deep copy a class instance via assignment result in a compile-time error.

Having defined a `clone` member function, we can make a deep copy of a `TimeOfDayI` instance as follows:

```
TimeOfDayIPtr tod1 = new TimeOfDayI;
TimeOfDayIPtr tod2 = tod1->clone();           // Deep copy
```

This results in the situation shown in Figure 6.8.

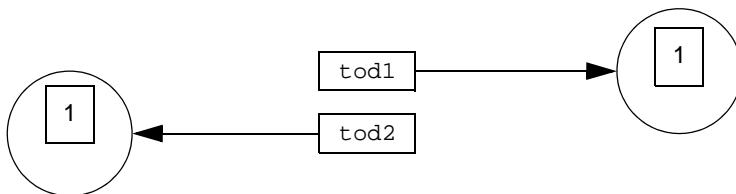


Figure 6.8. Deep-copying a class instance with `clone`.

Note that some older compilers still do not implement covariant return types. This causes problems if you have class hierarchies and want a `clone` member function at each level of the hierarchy. In that case, the only option is to make the return type of the `clone` member function a smart pointer to the base class, and, for all derived classes, to add static member function that can do a down cast. For example, given Slice classes `Base` and `Derived`, you can implement a `clone` method for `BaseI` and `DerivedI` as follows:

```
class BaseI;
typedef IceUtil::Handle<BaseI> BaseIPtr;

class BaseI : virtual public Base {
public:
    virtual BaseIPtr clone();
    // ...
};

class DerivedI;
typedef IceUtil::Handle<DerivedI> DerivedIPtr;
```

```
class DerivedI : virtual public Derived {
public:
    virtual BaseIPtr clone(); // Note: returns BaseIPtr
    // ...
};
```

Note that `DerivedI::clone` returns a `BaseIPtr` in this case (because we assume that the compiler cannot handle covariant return types). To make a deep copy of a `DerivedI` instance then, you can write:

```
DerivedIPtr p1 = new DerivedI;
DerivedIPtr p2 = DerivedIPtr::dynamicCast(p1->clone());
```

The `dynamicCast` static member function is part of the `IceUtil::Handle` template and, as the name suggests, implements a down-cast for smart pointers. As for a C++ `dynamic_cast`, the return value of `dynamicCast` is null if the argument is not of the expected type.

Null Smart Pointers

A null smart pointer contains a null C++ pointer to its underlying instance. This means that if you attempt to dereference a null smart pointer, you get an `IceUtil::NullHandleException`:

```
TimeOfDayPtr tod; // Construct null handle

bool gotNullHandleException = false;

try {
    tod->minute = 0; // Dereference null handle
} catch (const IceUtil::NullHandleException &) {
    gotNullHandleException = true;
}

assert(gotNullHandleException); // Must have seen exception
```

Stack-Allocated Class Instances

Despite the fact that class instances are reference-counted, you can still allocate a class instance on the stack without problems, for example:

```
{
    // Enter scope
    TimeOfDayI t; // Stack-allocated class instance
    // ...
} // Close scope, t is destroyed
```

When the thread of control leaves the block enclosing the variable `t`, the destructor of `t` is called by the compiler, which simply cleans up any dynamic resources that may be used by the class as usual. Nothing here calls `delete` because no smart pointer is involved. (Remember, it is the destructor of the *smart pointer* that calls `delete` when the reference count drops to zero, not the destructor of the class.)

Despite this, allocating class instances on the stack is pragmatically useless because all the Ice APIs expect parameters that are smart pointers, not class instances. This means that, to do anything with a stack-allocated class instance, you must initialize a smart pointer for the instance. However, doing so does not work because it inevitably leads to a crash:

```
{                                     // Enter scope
    TimeOfDayI t;                   // Stack-allocated class instance
    TimeOfDayPtr todp;              // Handle for a TimeOfDay instance

    todp = &t;                     // Legal, but dangerous
    // ...
}
```

// Leave scope, looming crash!

This goes badly wrong because, when `todp` goes out of scope, it decrements the reference count of the class to zero and calls `delete` on the stack-allocated class instance.

The following attempt to fix this is also doomed to failure:

```
{                                     // Enter scope
    TimeOfDayI t;                   // Stack-allocated class instance
    TimeOfDayPtr todp;              // Handle for a TimeOfDay instance

    todp = &t;                     // Legal, but dangerous
    // ...
    todp = 0;                      // Crash imminent!
}
```

This code attempts to circumvent the problem by clearing the smart pointer explicitly. However, doing so also causes the smart pointer to drop the reference count on the class to zero, so this code ends up with the same call to `delete` on the stack-allocated instance as the previous example.

The upshot of all this is: *never* allocate a class instance on the stack. The C++ mapping assumes that all class instances are allocated on the heap and no amount of coding trickery will change this.

Smart Pointers and Cycles

One thing you need to be aware of is the inability of reference counting to deal with cyclic dependencies. For example, consider the following simple self-referential class:

```
class Node {  
    int val;  
    Node next;  
};
```

Intuitively, this class implements a linked list of nodes. As long as there are no cycles in the list of nodes, everything is fine, and our smart pointers will correctly deallocate the class instances. However, if we introduce a cycle, we have a problem:

```
{                                // Open scope...  
  
    NodePtr n1 = new Node; // N1 refcount == 1  
    NodePrt n2 = new Node; // N2 refcount == 1  
    n1->next = n2;        // N2 refcount == 2  
    n2->next = n1;        // N1 refcount == 2  
  
} // destructors run:           // N2 refcount == 1,  
   // N1 refcount == 1, memory leak!
```

The nodes pointed to by `n1` and `n2` do not have names but, for the sake of illustration, let us assume that `n1`'s node is called N1, and `n2`'s node is called N2. When we allocate the N1 instance and assign it to `n1`, the smart pointer `n1` increments N1's reference count to 1. Similarly, N2's reference count is 1 after allocating the node and assigning it to `n2`. The next two statements set up a cyclic dependency between `n1` and `n2` by making their `next` pointers point at each other. This sets the reference count of both N1 and N2 to 2. When the enclosing scope closes, the destructor of `n2` is called first and decrements N2's reference count to 1, followed by the destructor of `n1`, which decrements N1's reference count to 1. The net effect is that neither reference count ever drops to zero, so both N1 and N2 are leaked.

Garbage Collection of Class Instances

The previous example illustrates a problem that is generic to using reference counts for deallocation: if a cyclic dependency exists anywhere in a graph (possibly via many intermediate nodes), all nodes in the cycle are leaked.

To avoid memory leaks due to such cycles, Ice for C++ contains a garbage collector. The collector identifies class instances that are part of one or more cycles but are no longer reachable from the program and deletes such instances:

- By default, garbage is collected whenever you destroy a communicator. This means that no memory is leaked when your program exits. (Of course, this assumes that you correctly destroy your communicators as described in Section 10.3.)
- You can also explicitly call the garbage collector by calling `Ice::collectGarbage`. For example, the leak caused by the preceding example can be avoided as follows:

```
{                                // Open scope...

    NodePtr n1 = new Node; // N1 refcount == 1
    NodePtr n2 = new Node; // N2 refcount == 1
    n1->next = n2;        // N1 refcount == 2
    n2->next = n1;        // N2 refcount == 2

} // Destructors run:           // N2 refcount == 1,
   // N1 refcount == 1

Ice::collectGarbage();          // Deletes N1 and N2
```

The call to `Ice::collectGarbage` deletes the no longer reachable instances N1 and N2 (as well as any other non-reachable instances that may have accumulated earlier).

- Deleting leaked memory with explicit calls to the garbage collector can be inconvenient because it requires polluting the code with calls to the collector. You can ask the Ice run time to run a garbage collection thread that periodically cleans up leaked memory by setting the property `Ice.GC.Interval` to a non-zero value.⁵ For example, setting `Ice.GC.Interval` to 5 causes the collector thread to run the garbage collector once every five seconds. You can trace the execution of the collector by setting `Ice.Trace.GC` to a non-zero value (see Appendix C).

Note that the garbage collector is useful *only* if your program actually creates cyclic class graphs. There is no point in running the garbage collector in programs that do not create such cycles. (For this reason, the collector thread is disabled by

5. See Chapter 14 for how to set properties.

default and runs only if you explicitly set `Ice.GC.Interval` to a non-zero value.)

Smart Pointer Comparison

As for proxy handles (see Section 6.11.3 on page 161), class handles support the comparison operators `==`, `!=`, and `<`. This allows you to use class handles in STL sorted containers. Be aware that, for smart pointers, object identity is *not* used for the comparison, because class instances do not have identity. Instead, these operators simply compare the memory address of the classes they point to. This means that `operator==` returns true only if two smart pointers point at the same physical class instance:

```
// Create a class instance and initialize
//
TimeOfDayIPtr p1 = new TimeOfDayI;
p1->hour = 23;
p1->minute = 10;
p1->second = 18;

// Create another class instance with
// the same member values
//
TimeOfDayIPtr p2 = new TimeOfDayI;
p2->hour = 23;
p2->minute = 10;
p2->second = 18;

assert(p1 != p2);           // The two do not compare equal

TimeOfDayIPtr p3 = p1;      // Point at first class again

assert(p1 == p3);           // Now they compare equal
```

6.15 slice2cpp Command-Line Options

The Slice-to-C++ compiler, `slice2cpp`, offers the following command-line options in addition to the standard options described in Section 4.18:

- **--header-ext EXT**

Changes the file extension for the generated header files from the default `h` to the extension specified by `EXT`.

- **--source-ext EXT**

Changes the file extension for the generated source files from the default `cpp` to the extension specified by `EXT`.

- **--impl**

Generate sample implementation files. This option will not overwrite an existing file.

- **--depend**

This option prints file dependencies on `stdout` in a form suitable for the `make` utility.

6.16 A Comparison with the CORBA C++ Mapping

Comparing the Slice and the CORBA C++ mappings is somewhat difficult because they are so different. As any CORBA C++ developer will know, the CORBA C++ mapping is large and complex and, in places, arcane. For example, the developer is burdened with a large number of error-prone memory management responsibilities, and the rules for what must be deallocated by the developer and what is deallocated by the ORB are inconsistent.

Overall, the Ice C++ mapping is much easier to use, integrates with STL and, due to the smaller amount of generated code, much more efficient.

Chapter 7

Developing a File System Client in C++

7.1 Chapter Overview

In this chapter, we present the source code for a C++ client that accesses the file system we developed in Chapter 5 (see Chapter 9 for the Java version of the client).

7.2 The C++ Client

We now have seen enough of the client-side C++ mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;
```

```
interface File extends Node {
    nonmutating Lines read();
    idempotent void write(Lines text) throws GenericError;
};

sequence<Node*> NodeSeq;

interface Directory extends Node {
    nonmutating NodeSeq list();
};

interface Filesyst {
    nonmutating Directory* getRoot();
};
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
#include <Ice/Ice.h>
#include <Filesystem.h>
#include <iostream>
#include <iterator>

using namespace std;
using namespace Filesystem;

static void
listRecursive(const DirectoryPrx & dir, int depth = 0)
{
    // ...
}

int
main(int argc, char * argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        // Create a communicator
        //
        ic = Ice::initialize(argc, argv);
```

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base
    = ic->stringToProxy("RootDir:default -p 10000");
if (!base)
    throw "Could not create proxy";

// Down-cast the proxy to a Directory proxy
//
DirectoryPrx rootDir = DirectoryPrx::checkedCast(base);
if (!rootDir)
    throw "Invalid proxy";

// Recursively list the contents of the root directory
//
cout << "Contents of root directory:" << endl;
listRecursive(rootDir);

} catch (const Ice::Exception & ex) {
    cerr << ex << endl;
    status = 1;
} catch (const char * msg) {
    cerr << msg << endl;
    status = 1;
}

// Clean up
//
if (ic)
    ic->destroy();

return status;
}
```

1. The code includes a few header files:

1. `Ice/Ice.h`

This file is always included in both client and server source files. It provides definitions that are necessary for accessing the Ice run time.

2. `Filesystem.h`

This is the header that is generated by the Slice compiler from the Slice definitions in `Filesystem.ice`.

3. `iostream`

The client uses the `iostream` library to produce its output.

4. iterator

The implementation of `listRecursive` uses an STL iterator.

2. The code adds using declarations for the `std` and `Filesystem` namespaces.
3. The structure of the code in `main` follows what we saw in Chapter 3. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
4. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`:

```
// Recursively print the contents of directory "dir" in
// tree fashion. For files, show the contents of each file.
// The "depth" parameter is the current nesting level
// (for indentation).

static void
listRecursive(const DirectoryPrx & dir, int depth = 0)
{
    string indent(++depth, '\t');

    NodeSeq contents = dir->list();

    for (NodeSeq::const_iterator i = contents.begin();
         i != contents.end();
         ++i) {
        DirectoryPrx dir = DirectoryPrx::checkedCast(*i);
        FilePrx file = FilePrx::uncheckedCast(*i);
        cout << indent << (*i)->name()
            << (dir ? " (directory):" : " (file):") << endl;
        if (dir) {
            listRecursive(dir, depth);
        } else {
            Lines text = file->read();
            for (Lines::const_iterator j = text.begin();
                 j != text.end();
                 ++j) {
                cout << indent << "\t" << *j << endl;
            }
        }
    }
}
```

```
        }
    }
}
```

The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the `Node` *is-a* `File` and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.

In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.

2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints "(directory)" or "(file)" following the name.
3. The code checks the type of the node:
 - If it is a directory, the code recurses, incrementing the indent level.
 - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:

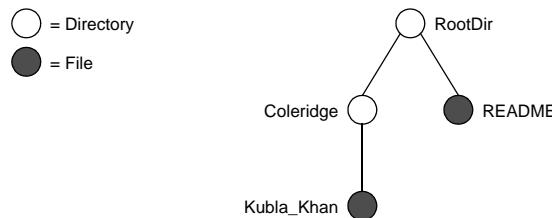


Figure 7.1. A small file system.

The output produced by client for this file system is:

```
Contents of root directory:
    README (file):
        This file system contains a collection of poetry.
    Coleridge (directory):
        Kubla_Khan (file):
            In Xanadu did Kubla Khan
            A stately pleasure-dome decree:
            Where Alph, the sacred river, ran
            Through caverns measureless to man
            Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in XREF and XREF.

7.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the C++ code hardly differs from the code you would write for an ordinary C++ program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local C++ object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to

struggle with arcane networking APIs. As we will see in Chapter 11, this is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

Chapter 8

Client-Side Slice-to-Java Mapping

8.1 Chapter Overview

In this chapter, we present the basic Slice-to-Java mapping (see Chapter 6 for the basic Slice-to-C++ mapping). One part of the client-side Java mapping concerns itself with rules for representing each Slice data type as a corresponding Java type; we cover these rules in Section 8.3 to Section 8.10. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Section 8.11 to Section 8.13. Slice classes have the characteristics of both data types and interfaces and are covered in Section 8.14. Finally, we conclude the chapter with a brief comparison of the Slice-to-Java mapping with the CORBA Java mapping.

8.2 Introduction

The client-side Slice-to-Java mapping defines how Slice data types are translated to Java types, and how clients invoke operations, pass parameters, and handle errors. Much of the Java mapping is intuitive. For example, Slice sequences map to Java arrays, so there is essentially nothing new you have learn in order to use Slice sequences in Java.

The Java API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data—the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Section 8.9 to Section 8.13 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the Java mapping, you should need no more than the Slice definition of your application and knowledge of the Java mapping rules. In particular, looking through the generated code in order to discern how to use the Java mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

8.3 Mapping for Identifiers

Slice identifiers map to an identical Java identifier. For example, the Slice identifier `Clock` becomes the Java identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a Java keyword, the corresponding Java identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.¹

1. As suggested in Section 4.5.3 on page 60, you should try to avoid such identifiers as much as possible.

8.4 Mapping for Modules

Slice modules map to Java packages with the same name as the Slice module. Definitions at the global scope become part of the unnamed Java global package. The mapping preserves the nesting of the Slice definitions. For example:

```
// Definitions at global scope here...

module M1 {
    // Definitions for M1 here...
    module M2 {
        // Definitions for M2 here...
    };
}

// ...

module M1 {      // Reopen M1
    // More definitions for M1 here...
};
```

This definition maps to the corresponding Java definitions:

```
// Definitions at global scope here...

package M1;
// Definitions for M1 here...

package M1.M2;
// Definitions for M2 here...

package M1;
// Definitions for M1 here...
```

Note that these definitions appear in the appropriate source files; source files for definitions at the global scope are generated in the top-level output directory, source files for definitions in module M1 are generated in directory M1 underneath the top-level directory, and source files for definitions for module M2 are generated in directory M1/M2 underneath the top-level directory. You can set the top-level output directory using the **--output-dir** option with **slice2java** (see Section 4.18).

8.5 The Ice Package

All of the APIs for the Ice run time are nested in the `Ice` package, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` package are generated from Slice definitions; other parts of the `Ice` package provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` package throughout the remainder of the book.

8.6 Mapping for Simple Built-In Types

The Slice built-in types are mapped to Java types as shown in Table 8.1.

Table 8.1. Mapping of Slice built-in types to Java.

Slice	Java
<code>bool</code>	<code>boolean</code>
<code>byte</code>	<code>byte</code>
<code>short</code>	<code>short</code>
<code>int</code>	<code>int</code>
<code>long</code>	<code>long</code>
<code>float</code>	<code>float</code>
<code>double</code>	<code>double</code>
<code>string</code>	<code>String</code>

8.7 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.


```

switch (f2.value()) {                                // Switch on enum
case Fruit._Apple:
    // ...
    break;
case Fruit._Pear
    // ...
    break;
case Fruit._Orange
    // ...
    break;
}

```

As you can see, the generated class enables natural use of enumerated values. The `int` members with a prepended underscore are constants that encode each enumerator; the `Fruit` members are preinitialized enumerators that you can use for initialization and comparison.

The `value` and `convert` methods act as an accessor and a modifier, so you can read and write the value of an enumerated variable as an `int`. If you are using the `convert` method, you must make sure that the passed value is within the range of the enumeration; failure to do so will result in an assertion failure:

```
Fruit favoriteFruit = Fruit.convert(4); // Assertion failure!
```

8.7.2 Mapping for Structures

Slice structures map to Java classes with the same name. For each Slice data member, the Java class contains a corresponding public data member. For example, here is our `Employee` structure from Section 4.7.4 once more:

```

struct Employee {
    long number;
    string firstName;
    string lastName;
};

```

The Slice-to-Java compiler generates the following definition for this structure:

```

public final class Employee implements java.lang.Cloneable {
    public long number;
    public String firstName;
    public String lastName;

    public boolean equals(java.lang.Object rhs) {
        // ...
    }
}

```

```
        }
    public int hashCode() {
        // ...
    }

    public java.lang.Object clone()
        throws java.lang.CloneNotSupportedException {
        return super.clone();
    }
}
```

For each data member in the Slice definition, the Java class contains a corresponding public data member of the same name. The `equals` member function compares two structures for equality. Note that the generated class also provides the usual `hashCode` and `clone` methods. (`clone` has the default behavior of making a shallow copy.)

8.7.3 Mapping for Sequences

Slice sequences map to Java arrays. This means that the Slice-to-Java compiler does not generate a separate named type for a Slice sequence. For example:

```
sequence<Fruit> FruitPlatter;
```

This definition simply corresponds to the Java type `Fruit[]`. Naturally, because Slice sequences are mapped to Java arrays, you can take advantage of all the array functionality provided by Java, such as initialization, assignment, cloning, and the `length` member. For example:

```
Fruit[] platter = { Fruit.Apple, Fruit.Pear };
assert(platter.length == 2);
```

Alternative Mappings

You can change the default mapping for sequences with a metadata directive, for example:

```
["java:type:java.util.LinkedList"] sequence<Fruit> FruitPlatter;
```

This instructs the compiler to use the type `java.util.LinkedList` in the generated code instead of using the default mapping to an array. In addition to using one of the collections provided by Java, you can also map sequences to your own custom implementation. As far as the Ice marshaling code is concerned, the

expectation is that, whatever type you use, it implements the `java.util.List` interface.

By adding a metadata directive to the sequence definition, you are changing the default mapping for that sequence type. You can also override how a particular structure, class, or exception member is mapped with additional metadata directives. For example:

```
enum Fruit { Apple, Pear, Orange };

sequence<Fruit> Breakfast;
["java:type:java.util.LinkedList"] sequence<Fruit> Dessert;

struct Meal1 {
    Breakfast   b;
    Dessert     d;
};

struct Meal2 {
    ["java:type:java.util.LinkedList"] Breakfast b;
    ["java:type:java.util.Vector"]      Dessert   d;
};
```

With this definition, `Breakfast` is mapped to a Java array, whereas `Dessert` is mapped to `java.util.LinkedList`, and the two data members of `Meal1` are mapped accordingly. For `Meal2`, the default mapping is overridden, so `Meal2::b` is mapped to `java.util.LinkedList` and `Meal2::d` is mapped to `java.util.Vector`.

Note that you can override the sequence mapping in this fashion only for structure, class, or exception members. (You cannot override the mapping for sequence elements, dictionary values, parameters, or return values.)

8.7.4 Mapping for Dictionaries

Here is the definition of our `EmployeeMap` from Section 4.7.4 once more:

```
dictionary<long, Employee> EmployeeMap;
```

As for sequences, the Java mapping does not create a separate named type for this definition. Instead, *all* dictionaries are simply of type `java.util.Map`, so we can use a map of employee structures like any other Java map. (Of course, because `java.util.Map` is an abstract class, we must use a concrete class, such as `java.util.HashMap` for the actual map.) For example:

```
java.util.Map em = new java.util.HashMap();

Employee e = new Employee();
e.number = 31;
e.firstName = "James";
e.lastName = "Gosling";

em.put(new Long(e.number), e);
```

8.8 Mapping for Constants

Here are the constant definitions we saw in Section 4.7.5 on page 69 once more:

```
const bool AppendByDefault = true;
const byte LowerNibble = 0x0f;
const string Advice = "Don't Panic!";
const short TheAnswer = 42;
const double PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

```
public interface AppendByDefault {
    boolean value = true;
}

public interface LowerNibble {
    byte value = 15;
}

public interface Advice {
    String value = "Don't Panic!";
}

public interface TheAnswer {
    short value = 42;
}

public interface PI {
    double value = 3.1416;
}
```

```
public interface FavoriteFruit {  
    Fruit value = Fruit.Pear;  
}
```

As you can see, each Slice constant is mapped to a Java interface with the same name as the constant. The interface contains a member named `value` that holds the value of the constant.

8.9 Mapping for Exceptions

Here is a fragment of the Slice definition for our world time server from Section 4.8.5 on page 85 once more:

```
exception GenericError {  
    string reason;  
};  
exception BadTimeVal extends GenericError {};  
exception BadZoneName extends GenericError {};
```

These exception definitions map as follows:

```
public class GenericError extends Ice.UserException {  
    public String reason;  
  
    public String ice_name() {  
        return "GenericError";  
    }  
}  
  
public class BadTimeVal extends GenericError {  
    public String ice_name() {  
        return "BadTimeVal";  
    }  
}  
  
public class BadZoneName extends GenericError {  
    public String ice_name() {  
        return "BadZoneName";  
    }  
}
```

Each Slice exception is mapped to a Java class with the same name. For each exception member, the corresponding class contains a public data member. (Obvi-

ously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception also defines the `ice_name` member function, which returns the name of the exception.

All user exceptions are derived from the base class `Ice.UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice.UserException`. `Ice.UserException`, in turn, derives from `java.lang.Exception`.

Note that the generated exception classes contain other member functions that are not shown. However, those member functions are internal to the Java mapping and are not meant to be called by application code.

8.10 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice.LocalException` (which, in turn, derives from `java.lang.RuntimeException`).

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 82. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice.LocalException`

This is the root of the inheritance tree for run-time exceptions.

- `Ice.UserException`

This is the root of the inheritance tree for user exceptions.

- `Ice.TimeoutException`

This is the base exception for both operation-invocation and connection-establishment timeouts.

- `Ice.ConnectionTimeoutException`

This exception is raised when the initial attempt to establish a connection to a server times out.

You will probably have little need to catch the remaining exceptions by category; the fine-grained error handling offered by the remainder of the hierarchy is of

interest mainly in the implementation of the Ice run time. However, there is one exception you will probably be interested in specifically:

`Ice.ObjectNotExistException`. This exception is raised if a client invokes an operation on an Ice object that no longer exists. In other words, the client holds a dangling reference to an object that probably existed some time in the past but has since been permanently destroyed.

8.11 Mapping for Interfaces

Slice interfaces map to proxies on the client side. A proxy is simply a Java interface with operations that correspond to the operations defined in the Slice interface.

The compiler generates quite few source files for each Slice interface. In general, for an interface `<interface-name>`, the following source files are created by the compiler:

- `<interface-name>.java`

This source file declares the `<interface-name>` Java interface.

- `<interface-name>Holder.java`

This source file defines a holder type for the interface (see page 216).

- `<interface-name>Prx.java`

This source file defines the `<interface-name>Prx` interface (see page 209).

- `<interface-name>PrxHelper.java`

This source file defines the helper type for the interface's proxy (see page 212).

- `<interface-name>PrxHolder.java`

This source file defines the a holder type for the interface's proxy (see page 216).

- `<interface-name>Operations.java`

This source file defines an interface that contains the operations corresponding to the Slice interface.

These are the files that contain code that is relevant to the client side. The compiler also generates a file that is specific to the server side, plus three additional files:

- `_<interface-name>Disp.java`

This file contains the definition of the server-side skeleton class.

- `_<interface-name>Del.java`
- `_<interface-name>DelD.java`
- `_<interface-name>DelM.java`

These files contain code that is internal to the Java mapping; they do not contain any functions of relevance to application programmers.

8.11.1 Proxy Interfaces

On the client side, Slice interfaces map to Java interfaces with member functions that correspond to the operations on those interfaces. Consider the following simple interface:

```
interface Simple {  
    void op();  
};
```

The Slice compiler generates the following definition for use by the client:

```
public interface SimplePrx extends Ice.ObjectPrx {  
    public void op();  
    public void op(java.util.Map __context);  
}
```

As you can see, the compiler generates a *proxy interface* `SimplePrx`. In general, the generated name is `<interface-name>Prx`. If an interface is nested in a module `M`, the generated class is part of package `M`, so the fully-qualified name is `M.<interface-name>Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a member function of the same name. For the preceding example, we find that the operation `op` has been mapped to the member function `op`. Also note that `op` is overloaded: the second version of `op` has a parameter `__context` of type `java.util.Map`. This parameter is for use by the Ice run time to store information about how to deliver a

request. You normally do not need to use it. (We examine the `__context` parameter in detail in Chapter 16. The parameter is also used by IceStorm—see Chapter 27.)

Because all the `<interface-name>Prx` types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. The proxy references handed out by the Ice run time are always of type `<interface-name>Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

8.11.2 The `Ice.ObjectPrx` Interface

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice.ObjectPrx`. `ObjectPrx` provides a number of methods:

```
package Ice;

public interface ObjectPrx {
    boolean equals(java.lang.Object r);
    Identity ice_getIdentity();
    int ice_hash();
    boolean ice_isA(String __id);
    String ice_id();
    void ice_ping();
    // ...
}
```

The methods behave as follows:

- `equals`

This operation compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

- `ice_getIdentity`

This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

```
module Ice {  
    struct Identity {  
        string name;  
        string category;  
    };  
};
```

To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

```
Ice.ObjectPrx o1 = ...;  
Ice.ObjectPrx o2 = ...;  
Ice.Identity i1 = o1.ice_getidentity();  
Ice.Identity i2 = o2.ice_getidentity();  
  
if (i1.equals(i2))  
    // o1 and o2 denote the same object  
else  
    // o1 and o2 denote different objects
```

- **ice_hash**

This method returns a hash key for the proxy.

- **ice_isA**

This method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID (see Section 4.12). For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

```
Ice.ObjectPrx o = ...;  
if (o != null && o.ice_isA("::Printer"))  
    // o denotes a Printer object  
else  
    // o denotes some other type of object
```

Note that we are testing whether the proxy is null before attempting to invoke the `ice_isA` method. This avoids getting a `NullPointerException` if the proxy is null.

- **ice_id**

This method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type `BasePrx`, with a static type ID of `::Base`, the return value of `ice_id` might be `::Base`, or it might something more derived, such as `::Derived`.

- `ice_ping`

This method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as

`ObjectNotExistException` or `ConnectTimeoutException`.

Note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call. (We discuss these methods in Chapter 16.)

8.11.3 Proxy Helpers

For each Slice interface, apart from the proxy interface, the Slice-to-Java compiler creates a helper class: for an interface `Simple`, the name of the generated helper class is `SimplePrxHelper`. The helper classes contains two methods of interest:

```
public final class SimplePrxHelper
    extends Ice.ObjectPrxHelper implements SimplePrx {
    public static SimplePrx checkedCast(Ice.ObjectPrx b) {
        // ...
    }

    public static SimplePrx uncheckedCast(Ice.ObjectPrx b) {
        // ...
    }

    // ...
}
```

Both the `checkedCast` and `uncheckedCast` methods implement a down-cast: if the passed proxy is a proxy for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a non-null reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns a null reference.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

```
Ice.ObjectPrx obj = ...;           // Get a proxy from somewhere...

SimplePrx simple = SimplePrxHelper.checkedCast(obj);
if (simple != null)
    // Object supports the Simple interface...
else
    // Object is not of type Simple...
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of a proxy in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`. (This also explains the need for the helper class: the Ice run time must contact the server, so we cannot use a Java down-cast.)

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather non-sensical things. To illustrate this, consider the following two interfaces:

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

```
Ice.ObjectPrx obj = ...;           // Get proxy...
ProcessPrx process
    = ProcessPrxHelper::uncheckedCast(obj); // No worries...
process.launch(40, 60);            // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

A final warning about down-casts: you must use either a `checkedCast` or an `uncheckedCast` to down-cast a proxy. If you use a Java cast, the behavior is undefined.

8.12 Mapping for Operations

As we saw in Section 8.11, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system from Section 5.4:

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```
NodePrx node = ...;           // Initialize proxy
String name = node.name();    // Get name via RPC
```

This illustrates the typical pattern for receiving return values: return values are returned by reference for complex types, and by value for simple types (such as `int` or `double`).

8.12.1 Normal, idempotent, and nonmutating Operations

You can add an `idempotent` or `nonmutating` qualifier to a Slice operation. As far as the signature for the corresponding proxy methods is concerned, neither `idempotent` nor `nonmutating` have any effect. For example, consider the following interface:

```
interface Example {  
    string op1();  
    idempotent string op2();  
    nonmutating string op3();  
};
```

The proxy interface for this is:

```
public interface ExamplePrx extends Ice.ObjectPrx {  
    public String op1();  
    public String op2();  
    public String op3();  
}
```

`idempotent` and `nonmutating` affect an aspect of call dispatch, not interface, so it makes sense for the three methods to look the same.

8.12.2 Passing Parameters

In-Parameters

The parameter passing rules for the Java mapping are very simple: parameters are passed either by value (for simple types) or by reference (for complex types and type `String`). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats—see XREF).

Here is an interface with operations that pass parameters of various types from client to server:

```
struct NumberAndString {  
    int x;  
    string str;  
};  
  
sequence<string> StringSeq;  
  
dictionary<long, StringSeq> StringTable;
```

```
interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following proxy for this definition:

```
public interface ClientToServerPrx extends Ice.ObjectPrx {
    public void op1(int i, float f, boolean b, String s);
    public void op2(NumberAndString ns,
                   String[] ss,
                   java.util.Map st);
    public void op3(ClientToServerPrx proxy);
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```
ClientToServerPrx p = ...;                                // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
boolean b = true;
String s = "Hello world!";
p.op1(i, f, b, s);                                     // Pass simple variables

NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
String[] ss = { "Hello world!" };
java.util.HashMap st = new java.util.HashMap();
st.put(new Long(0), ns);
p.op2(ns, ss, st);                                      // Pass complex variables

p.op3(p);                                              // Pass proxy
```

out-Parameters

Java does not have pass-by-reference: parameters are always passed by value. For a function to modify one of its arguments, we must pass a reference (by value) to an object; the called function can then modify the object's contents via the passed reference.

To permit the called function to modify a parameter, the Java mapping provides so-called *holder* classes. For example, for each of the built-in Slice types, such as `int` and `string`, the `Ice` package contains a corresponding holder class. Here are the definitions for the holder classes `Ice.IntHolder` and `Ice.StringHolder`:

```
package Ice;

public final class IntHolder {
    public IntHolder() {}
    public IntHolder(int value)
        this.value = value;
    }
    public int value;
}

public final class StringHolder {
    public StringHolder() {}
    public StringHolder(String value) {
        this.value = value;
    }
    public String value;
}
```

A holder class has a public `value` member that stores the value of the parameter; the called function can modify the value by assigning to that member. The class also has a default constructor and a constructor that accepts an initial value.

For user-defined types, such as structures, the Slice-to-Java compiler generates a corresponding holder type. For example, here is the generated holder type for the `NumberAndString` structure we defined on page 215:

```
public final class NumberAndStringHolder {
    public NumberAndStringHolder() {}

    public NumberAndStringHolder(NumberAndString value) {
        this.value = value;
    }

    public NumberAndString value;
}
```

This looks exactly like the holder classes for the built-in types: we get a default constructor, a constructor that accepts an initial value, and the public `value` member.

Note that holder classes are generated for *every* Slice type you define. For example, for sequences, such as the `FruitPlatter` sequence we saw on page 203, the compiler does not generate a special Java `FruitPlatter` type because sequences map to Java arrays. However, the compiler *does* generate a `FruitPlatterHolder` class, so we can pass a `FruitPlatter` array as an `out`-parameter.

To pass an `out`-parameter to an operation, we simply pass an instance of a holder class and examine the `value` member of each `out`-parameter when the call completes. Here is the same Slice definition we saw on page 215 once more, but this time with all parameters being passed in the `out` direction:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
            out StringSeq ss,
            out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
public interface ClientToServerPrx extends Ice.ObjectPrx {
    public void op1(Ice.IntHolder i, Ice.FloatHolder f,
                   Ice.BooleanHolder b, Ice.StringHolder s);
    public void op2(NumberAndStringHolder ns,
                   StringSeqHolder ss, StringTableHolder st);
    public void op3(ClientToServerPrxHolder proxy);
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

```
ClientToServerPrx p = ...; // Get proxy...
Ice.IntHolder ih = new Ice.IntHolder();
Ice.FloatHolder fh = new Ice.FloatHolder();
```

```
Ice.BooleanHolder bh = new Ice.BooleanHolder();
Ice.StringHolder sh = new Ice.StringHolder();
p.op1(ih, fh, bh, sh);

NumberAndStringHolder nsh = new NumberAndString();
StringSeqHolder ssh = new StringSeqHolder();
StringTableHolder sth = new StringTableHolder();
p.op2(nsh, ssh, sth);

ServerToClientPrxHolder stcph = new ServerToClientPrxHolder();
p.op3(stcph);

System.out.writeln(ih.value); // Show one of the values
```

Again, there are no surprises in this code: the various holder instances contain values once the operation invocation completes and the `value` member of each instance provides access to those values.

Parameter Type Mismatches

Parameters in the Java mapping are statically type safe, with one exception: dictionaries map to `java.util.Map`, which is simply a mapping from `java.lang.Object` to `java.lang.Object`. It follows that, if you pass a map between client and server, you must take care to ensure that the pairs of values you insert into the map are of the correct type. For example:

```
dictionary<string, long> AgeTable;

interface Ages {
    void put(AgeTable ages);
}
```

Given a proxy to an `Ages` object, the client code could look as follows:

```
AgesPrx ages = ...; // Get proxy...

java.util.HashMap ageTable = new java.util.HashMap();
String name = "Michi Henning";
Long age = new Long(42);
ageTable.put(age, name); // Oops...
ages.put(ageTable); // ClassCastException!
```

The problem here is that the order of the parameters for `ageTable.put` is reversed. When the proxy implementation tries to marshal the data, it notices the type mismatch and throws a `ClassCastException`.

Null Parameters

Some Slice types naturally have “empty” or “not there” semantics. Specifically, proxies, sequences, dictionaries, and strings all can be null:

- For proxies, a Java null references denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points “nowhere” (denotes no object).
- Sequences and dictionaries cannot be null, but can be empty. To make life with these types easier, whenever you pass a Java null reference as a parameter or return value of type sequence or dictionary, the Ice run time automatically sends an empty sequence or dictionary to the receiver.
- Java strings can be null, but Slice strings cannot (because Slice strings do not support the concept of a null string). Whenever you pass a Java null string as a parameter or return value, the Ice run time automatically sends an empty string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending it in order to avoid `NullPointerException`s. Note that using null parameters in this way does *not* create null semantics for sequences, dictionaries, or string. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example if you send a string as a null reference or an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

8.13 Exception Handling

Any operation invocation may throw a run-time exception (see Section 8.10 on page 207) and, if the operation has an exception specification, may also throw user exceptions (see Section 8.9 on page 206). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as Java exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

```
ChildPrx child = ...; // Get child proxy...

try {
    child.askToCleanUp();
} catch (Tantrum t) {
    System.out.write("The child says: ");
    System.out.writeln(t.reason);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

```
public class Client {
    static void run() {
        ChildPrx child = ...; // Get child proxy...
        try {
            child.askToCleanUp();
        } catch (Tantrum t) {
            System.out.print("The child says: ");
            System.out.println(t.reason);
            p.scold(); // Recover from error...
        }
        p.praise(); // Give positive feedback...
    }

    public static void
    main(String[] args)
    {
        try {
            // ...
            run();
            // ...
        } catch (Ice.LocalException e) {
            e.printStackTrace();
        }
    }
}
```

```

        } catch (Ice.UserException e) {
            System.err.println(e.getMessage());
        }
    }
}

```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 3.)

Exceptions and out-Parameters

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [19] but does not provide the strong exception guarantee.²

8.14 Mapping for Classes

Slice classes are mapped to Java classes with the same name. The generated class contains a public data member for each Slice data member (just as for structures and exceptions), and a member function for each operation. Consider the following class definition:

```

class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 -59
    string format();      // Return time as hh:mm:ss
};

```

The Slice compiler generates the following code for this definition:

```

public interface TimeOfDayOperations {
    String format(Ice.Current current);
}

```

2. This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

```
public abstract class TimeOfDay extends Ice.ObjectImpl
    implements TimeOfDayOperations {
    public short hour;
    public short minute;
    public short second;
    // ...
}
```

Note the following about the generated code:

- The compiler generates an interface called `TimeOfDayOperations`. This `<interface-name>Operations` interface contains a method for each Slice operation in the class.

If a class has only data members, but no operations, the compiler omits generating the `<interface-name>Operations` class.

- The compiler generates an abstract base class with the same name as the Slice class (`TimeOfDay` in this example). This class contains a public data member for each data member of the Slice class.

If a class has only data members, but no operations, the compiler generates a non-abstract class.

In this example, the `TimeOfDay` class is abstract, which makes sense: the Slice-to-Java compiler has no way of knowing what the implementation of the `format` operation should do, so the only choice is to make `TimeOfDay` abstract. To create a working `TimeOfDay` implementation, you must derive a class from `TimeOfDay` that supplies the missing `format` implementation, for example:

```
public class TimeOfDayI extends TimeOfDay {
    public String format(Ice.Current current) {
        DecimalFormat df
            = (DecimalFormat)DecimalFormat.getInstance();
        df.setMinimumIntegerDigits(2);
        return new String(df.format(hour) + ":" +
                           df.format(minute) + ":" +
                           df.format(second));
    }
}
```

8.14.1 Class Factories

Having created a class such as this, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an

out-parameter from an operation invocation. To see why, consider the following simple interface:

```
interface Time {  
    TimeOfDay get();  
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

```
module Ice {  
    local interface ObjectFactory {  
        Object create(string type);  
        void destroy();  
    };  
  
    local interface Communicator {  
        void addObjectFactory(ObjectFactory factory, string id);  
        void removeObjectFactory(string id);  
        ObjectFactory findObjectFactory(string id);  
        // ...  
    };  
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

```
class ObjectFactory  
    extends Ice.LocalObjectImpl  
    implements Ice.ObjectFactory {  
public Ice.Object create(String type) {  
    if(type.equals(":TimeOfDay")) {  
        return new TimeOfDayI();  
    }  
    assert(false);  
    return null;  
}
```

```
    public void destroy() {
        // Nothing to do
    }
}
```

The object factory's `create` method is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` method is called by the Ice run time when the factory is unregistered or its `Communicator` is destroyed.

The `create` method is passed the type ID (see Section 4.12) of the class to instantiate. For our `TimeOfDay` class, the type ID is "`::TimeOfDay`". Our implementation of `create` checks the type ID: if it is "`::TimeOfDay`", it instantiates and returns a `TimeOfDayI` object. For other type IDs, it asserts because it does not know how to instantiate other types of objects.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
Ice.Communicator ic = ...;
ic.addObjectFactory(new ObjectFactory(), "::TimeOfDay");
```

Now, whenever the Ice run time needs to instantiate a class with the type ID "`::TimeOfDay`", it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when you call `Communicator::removeObjectFactory` or when the `Communicator` is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the `Communicator`—if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`. Similarly, attempting to remove the factory for a type ID that is not registered throws a `NotRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

8.14.2 Inheritance from LocalObject

Note that the implementation of the factory in the previous example extends `Ice.LocalObjectImpl`. The object factory, being a local object, inherits a number of operations from `Ice.LocalObject`, such as `ice_hash` (see page 211). The `Ice.LocalObjectImpl` class provides implementations for these operations, so you do not need to implement them yourself.

In general, all local interfaces inherit from `Ice.LocalObject`, and all implementations of local interfaces, such as an object factory or servant locator (see Section 16.6) must inherit from `Ice.LocalObjectImpl`.

8.15 Packages

By default, the scope of a Slice definition determines the package of its mapped Java construct. If a Slice type is defined outside of any module, it is mapped to a Java type residing in the unnamed global package. Similarly, a Slice type defined in a module hierarchy is mapped to a type residing in the equivalent Java package (see Section 8.4 for more information on the module mapping).

There are times, however, when applications require greater control over the packaging of generated Java classes. For instance, a company may have corporate software development guidelines requiring all Java classes to reside in a designated package. One way to satisfy this requirement is to modify the Slice module hierarchy so that the generated code uses the required package by default. In the example below, we have enclosed the original definition of `Workflow:Document` in the modules `com:acme` so that the compiler will create the class in the `com.acme` package:

```
module com {
    module acme {
        module Workflow {
            class Document {
                // ...
            };
        };
    };
}
```

There are two problems with this workaround:

1. It incorporates the requirements of an implementation language into the application's interface specification.

2. Developers using other languages, such as C++, are also affected.

The Slice-to-Java compiler provides a better way to control the packages of generated code through the use of global metadata (see Section 4.17). The example above can be converted as follows:

```
[["java:package:com.acme"]]
module Workflow {
    class Document {
        // ...
    };
}
```

The global metadata directive `java:package:com.acme` instructs the compiler to generate all of the classes resulting from definitions in this Slice file into the Java package `com.acme`. The net effect is the same: the class for `Document` is generated in the package `com.acme.Workflow`. However, we have addressed the two shortcomings of the first solution by reducing our impact on the interface specification: the Slice-to-Java compiler recognizes the package metadata directive and modifies its actions accordingly, whereas the compilers for other language mappings simply ignore it.

8.16 slice2java Command-Line Options

The Slice-to-Java compiler, **slice2java**, offers the following command-line options in addition to the standard options described in Section 4.18:

- **--tie**
Generate tie classes (see Section 12.7).
- **--impl**
Generate sample implementation files. This option will not overwrite an existing file.
- **--impl-tie**
Generate sample implementation files using ties (see Section 12.7). This option will not overwrite an existing file.

Chapter 9

Developing a File System Client in Java

9.1 Chapter Overview

In this chapter, we present the source code for a Java client that accesses the file system we developed in Chapter 5 (see Chapter 7 for the C++ version of the client).

9.2 The Java Client

We now have seen enough of the client-side Java mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
module Filesystem {  
    interface Node {  
        nonmutating string name();  
    };  
  
    exception GenericError {  
        string reason;  
    };  
  
    sequence<string> Lines;
```

```
interface File extends Node {
    nonmutating Lines read();
    idempotent void write(Lines text) throws GenericError;
};

sequence<Node*> NodeSeq;

interface Directory extends Node {
    nonmutating NodeSeq list();
};

interface Filesyst {
    nonmutating Directory* getRoot();
};
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
import Filesystem.*;

public class Client {

    // Recursively print the contents of directory "dir" in
    // tree fashion.  For files, show the contents of each file.
    // The "depth" parameter is the current nesting level
    // (for indentation).

    static void
    listRecursive(DirectoryPrx dir, int depth)
    {
        char[] indentCh = new char[++depth];
        java.util.Arrays.fill(indentCh, '\t');
        String indent = new String(indentCh);

        NodePrx[] contents = dir.list();

        for (int i = 0; i < contents.length; ++i) {
            DirectoryPrx subdir
                = DirectoryPrxHelper.checkedCast(contents[i]);
            FilePrx file
                = FilePrxHelper.uncheckedCast(contents[i]);
            if (subdir != null) {
                System.out.println(indent + "d " + contents[i].name());
                listRecursive(subdir, depth + 1);
            } else {
                System.out.println(indent + "f " + contents[i].name());
                System.out.println(indent + "c " + contents[i].read());
            }
        }
    }
}
```

```
        System.out.println(indent + contents[i].name() +
                           (subdir != null ? " (directory):" : " (file):"));
        if (subdir != null) {
            listRecursive(subdir, depth);
        } else {
            String[] text = file.read();
            for (int j = 0; j < text.length; ++j)
                System.out.println(indent + "\t" + text[j]);
        }
    }
}

public static void
main(String[] args)
{
    int status = 0;
    Ice.Communicator ic = null;
    try {
        // Create a communicator
        //
        ic = Ice.Util.initialize(args);

        // Create a proxy for the root directory
        //
        Ice.ObjectPrx base
            = ic.stringToProxy("RootDir:default -p 10000");
        if (base == null)
            throw new RuntimeException("Cannot create proxy");

        // Down-cast the proxy to a Directory proxy
        //
        DirectoryPrx rootDir
            = DirectoryPrxHelper.checkedCast(base);
        if (rootDir == null)
            throw new RuntimeException("Invalid proxy");

        // Recursively list the contents of the root directory
        //
        System.out.println("Contents of root directory:");
        listRecursive(rootDir, 0);
    } catch (Ice.LocalException e) {
        e.printStackTrace();
        status = 1;
    } catch (Exception e) {
        System.err.println(e.getMessage());
        status = 1;
```

```
        } finally {
            // Clean up
            //
            if (ic != null)
                ic.destroy();
        }

        System.exit(status);
    }
}
```

After importing the `Filesystem` package, the `Client` class defines two methods: `listRecursive`, which is a helper function to print the contents of the file system, and `main`, which is the main program. Let us look at `main` first:

1. The structure of the code in `main` follows what we saw in Chapter 3. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` is-a `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we know that the `Node` is-a `File` and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.

In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.

2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints "(directory)" or "(file)" following the name.
3. The code checks the type of the node:

- If it is a directory, the code recurses, incrementing the indent level.
- If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:

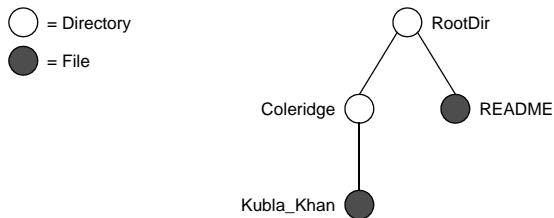


Figure 9.1. A small file system.

The output produced by client for this file system is:

Contents of root directory:

```
    README (file):
        This file system contains a collection of poetry.
    Coleridge (directory):
        Kubla_Khan (file):
            In Xanadu did Kubla Khan
            A stately pleasure-dome decree:
            Where Alph, the sacred river, ran
            Through caverns measureless to man
            Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in Chapter 21 and XREF.

9.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the Java code hardly

differs from the code you would write for an ordinary Java program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Java object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. As we will see in Chapter 13, this is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

Chapter 10

Server-Side Slice-to-C++ Mapping

10.1 Chapter Overview

In this chapter, we present the server-side Slice-to-C++ mapping (see Chapter 12 for the server-side Slice-to-Java mapping). Section 10.3 discusses how to initialize and finalize the server-side run time, sections 10.4 to 10.6 show how to implement interfaces and operations, and Section 10.7 discusses how to register objects with the server-side Ice run time.

10.2 Introduction

The mapping for Slice data types to C++ is identical on the client side and server side. This means that everything in Chapter 6 also applies to the server side. However, for the server side, there are a few additional things you need to know, specifically:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions
- how to create servants and register them with the Ice run time.

We discuss these topics in the remainder of this chapter.

10.3 The Server-Side main Function

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. As for the client side, you must initialize the Ice run time by calling `Ice::initialize` before you can do anything else in your server. `Ice::initialize` returns a smart pointer to an instance of an `Ice::Communicator`:

```
int
main(int argc, char * argv[])
{
    Ice::CommunicatorPtr ic
        = Ice::initialize(argc, argv);
    // ...
}
```

`Ice::initialize` accepts a C++ reference to `argc` and `argv`. The function scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `Ice::initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your `main` function, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` waits for any operation invocations that may still be running to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your `main` function to terminate without calling `destroy` first; doing so has undefined behavior.

The general shape of our server-side `main` function is therefore as follows:

```
#include <Ice/Ice.h>

int
main(int argc, char * argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);

        // Server code here...
    }
```

```
    } catch (const Ice::Exception & e) {
        cerr << e << endl;
        status = 1;
    } catch (const std::string & msg) {
        cerr << msg << endl;
        status = 1;
    } catch (const char * msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

Note that the code places the call to `Ice::initialize` in to a `try` block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

The `catch` handlers for `const std::string &` and `const char *` are in place as a convenience feature: if we encounter a fatal error condition anywhere in the server code, we can simply throw a string or a string literal containing an error message; this causes the stack to be unwound back to `main`, at which point the error message is printed and, after destroying the communicator, `main` terminates with non-zero exit status.

10.3.1 The `Ice::Application` Class

The preceding structure for the `main` function is so common that Ice offers a class, `Ice::Application`, that encapsulates all the correct initialization and finalization activities. The definition of the class is as follows (with some detail omitted for now):

```
namespace Ice {
    class Application /* ... */ {
public:
    Application();
    virtual ~Application();

    int main(int, char * [], const char * = 0);
    virtual int run(int, char * []) = 0;

    static const char * appName();
}
```

```
        static CommunicatorPtr communicator();
        // ...
    };
}
```

The intent of this class is that you specialize `Ice::Application` and implement the pure virtual `run` method in your derived class. Whatever code you would normally place in `main` goes into the `run` method instead. Using `Ice::Application`, our program looks as follows:

```
#include <Ice/Ice.h>

class MyApplication : virtual public Ice::Application {
public:
    virtual int run(int, char * []) {
        // Server code here...

        return 0;
    }
};

int
main(int argc, char * argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

The `Application::main` function does the following:

1. It installs an exception handler for `Ice::Exception`. If your code fails to handle an Ice exception, `Application::main` prints the exception details on `stderr` before returning with a non-zero return value.
2. It installs exception handlers for `const std::string &` and `const char *`. This allows you to terminate your server in response to a fatal error conditions by throwing a `std::string` or a string literal. `Application::main` prints the string on `stderr` before returning a non-zero return value.
3. It initializes (by calling `Ice::initialize`) and finalizes (by calling `Communicator::destroy`) a communicator. You can get access to the communicator for your server by calling the static `communicator()` member.

4. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.
5. It provides the name of your application via the static `appName` member function. The return value from this call is `argv[0]`, so you can get at `argv[0]` from anywhere in your code by calling `Ice::Application::appName` (which is usually required for error messages).
6. It creates an `IceUtil::CtrlCHandler` that properly shuts down the communicator.

Using `Ice::Application` ensures that your program properly finalizes the Ice run time, whether your server terminates normally or in response to an exception or signal. We recommend that all your programs use this class; doing so makes your life easier. In addition `Ice::Application` also provides features for signal handling and configuration that you do not have implement yourself when you use this class.

Using `Ice::Application` on the Client Side

You can use `Ice::Application` for your clients as well: simply implement a class that derives from `Ice::Application` and place the client code into its `run` method. The advantage of this approach is the same as for the server side: `Ice::Application` ensures that the communicator is destroyed correctly even in the presence of exceptions.

Catching Signals

The simple server we developed in Chapter 3 had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

To make it easier to deal with signals, `Ice::Application` encapsulates the platform-independent signal handling capabilities provided by the class `IceUtil::CtrlCHandler` (see Section 15.11). This allows you to cleanly

shut down on receipt of a signal and to use the same source code regardless of the underlying operating system and threading package:

```
namespace Ice {
    class Application : /* ... */ {
public:
    // ...
    static void shutdownOnInterrupt();
    static void ignoreInterrupt();
    static void holdInterrupt();
    static void releaseInterrupt();
    static bool interrupted();
};
}
```

You can use `Ice::Application` under both Windows and UNIX: for UNIX, the member functions control the behavior of your application for **SIGINT**, **SIGHUP**, and **SIGTERM**; for Windows, the member functions control the behavior of your application for **CTRL_C_EVENT**, **CTRL_BREAK_EVENT**, **CTRL_CLOSE_EVENT**, **CTRL_LOGOFF_EVENT**, and **CTRL_SHUTDOWN_EVENT**.

The member functions behave as follows:

- `shutdownOnInterrupt`

This function creates an `IceUtil::CtrlCHandler` that cleanly shuts down your application. This is the default behavior.

- `ignoreInterrupt`

This function causes signals to be ignored.

- `holdInterrupt`

This function temporarily blocks signal delivery.

- `releaseInterrupt`

This function restores signal delivery to the previous disposition. Any signal that arrives after `holdInterrupt` was called is delivered when you call `releaseInterrupt`.

- `interrupted`

This function returns `true` if a signal caused the communicator to shut down, `false` otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal. This is useful, for example, for logging purposes.

By default, `Ice::Application` behaves as if `shutdownOnInterrupt` was invoked, therefore our server `main` function requires no change to ensure that the program terminates cleanly on receipt of a signal. However, we add a diagnostic to report the occurrence of a signal, so our `main` function now looks like:

```
#include <Ice/Ice.h>

class MyApplication : virtual public Ice::Application {
public:
    virtual int run(int, char * []) {
        // Server code here...

        if (interrupted())
            cerr << appName() << ": terminating" << endl;

        return 0;
    }
};

int
main(int argc, char * argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

Note that, if your server is interrupted by a signal, the Ice run time waits for all currently executing operations to finish. This means that an operation that updates persistent state cannot be interrupted in the middle of what it was doing and cause partial update problems.

`Ice::Application` and Properties

Apart from the functionality shown in this section, `Ice::Application` also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or port number for a server. We discuss Ice properties in more detail in Chapter 14.

Limitations of `Ice::Application`

`Ice::Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use `Ice::Application`.

Instead, you must structure your code as we saw in Chapter 3 (taking care to always destroy the communicator).

10.3.2 The `Ice::Service` Class

The `Ice::Application` class described in Section 10.3.1 is very convenient for general use by Ice client and server applications. In some cases, however, an application may need to run at the system level as a Unix daemon or Win32 service. For these situations, Ice includes `Ice::Service`, a singleton class that is comparable to `Ice::Application` but also encapsulates the low-level, platform-specific initialization and shutdown procedures common to system services. The `Ice::Service` class is defined as follows:

```
namespace Ice {
    class Service {
public:
    Service();

    virtual bool shutdown();
    virtual void interrupt();

    int main(int, char * []);
    Ice::CommunicatorPtr communicator() const;

    static Service * instance();

    bool service() const;
    std::string name() const;
    bool checkSystem() const;

    int run(int, char * []);

    void configureService(const std::string &);

    int installService(const std::string &,
                      const std::string &,
                      const std::string &,
                      const std::vector<std::string> &);
    int uninstallService(const std::string &);
    int startService(const std::string &,
                    const std::vector<std::string> &);
    int stopService(const std::string &);

    void configureDaemon(bool, bool);
```

```
        virtual void handleInterrupt(int);

protected:
    virtual bool start(int, char * []) = 0;
    virtual void waitForShutdown();
    virtual bool stop();
    virtual Ice::CommunicatorPtr initializeCommunicator(
        int &, char * []);

    virtual void syserror(const std::string &);
    virtual void error(const std::string &);
    virtual void warning(const std::string &);
    virtual void trace(const std::string &);

    void enableInterrupt();
    void disableInterrupt();

    // ...
};

}
```

At a minimum, an Ice application that uses the `Ice::Service` class must define a subclass and override the `start` member function, which is where the service must perform its startup activities, such as processing command-line arguments, creating an object adapter, and registering servants. The application's main function must instantiate the subclass and typically invokes its main member function, passing the program's argument vector as parameters. The example below illustrates a minimal `Ice::Service` subclass:

```
#include <Ice/Ice.h>
#include <Ice/Service.h>

class MyService : public Ice::Service {
protected:
    virtual bool start(int, char * []);
private:
    Ice::ObjectAdapterPtr _adapter;
};

bool
MyService::start(int argc, char * argv[])
{
    _adapter = communicator()->createObjectAdapter("MyAdapter");
    _adapter->addWithUUID(new MyServantI);
```

```
    _adapter->activate();
    return true;
}

int
main(int argc, char * argv[])
{
    MyService svc;
    return svc.main(argc, argv);
}
```

The `Service::main` member function performs the following sequence of tasks:

1. Scans the argument vector for reserved options that indicate whether the program should run as a system service. Additional reserved options are supported for administrative tasks.
2. Configures the program for running as a system service (if necessary) by invoking `configureService` or `configureDaemon`, as appropriate for the platform.
3. Invokes the `run` member function and returns its result.

The `Service::run` member function executes the service in the steps shown below:

1. Installs an `IceUtil::CtrlCHandler` (see Section 15.11) for proper signal handling.
2. Invokes the `initializeCommunicator` member function to obtain a communicator. The communicator instance can be accessed using the `communicator` member function.
3. Invokes the `start` member function. If `start` returns `false` to indicate failure, `run` destroys the communicator and returns immediately.
4. Invokes the `waitForShutdown` member function, which should block until `shutdown` is invoked.
5. Invokes the `stop` member function. If `stop` returns `true`, `run` considers the application to have terminated successfully.
6. Destroys the communicator.
7. Gracefully terminates the system service (if necessary).

If an unhandled exception is caught by `Service::run`, a descriptive message is logged, the communicator is destroyed and the service is terminated.

Ice::Service Member Functions

The virtual member functions in `Ice::Service` represent the points at which a subclass can intercept the service activities. All of the virtual member functions (except `start`) have default implementations.

- `void handleInterrupt(int sig)`
Invoked by the `CtrlCHandler` when a signal occurs. The default implementation ignores the signal if it represents a logoff event and the `Ice.Nohup` property is set to a value larger than zero, otherwise it invokes the `interrupt` member function.
- `Ice::CommunicatorPtr initializeCommunicator(int & argc, char * argv[])`
Initializes a communicator. The default implementation invokes `Ice::initialize` and passes the given argument vector.
- `void interrupt()`
Invoked by the signal handler to indicate a signal was received. The default implementation invokes the `shutdown` member function.
- `bool shutdown()`
Causes the service to begin the shutdown process. The default implementation invokes `shutdown` on the communicator. The subclass must return `true` if shutdown was started successfully, and `false` otherwise.
- `bool start(int argc, char * argv[])`
Allows the subclass to perform its startup activities, such as scanning the provided argument vector for recognized command-line options, creating an object adapter, and registering servants. The subclass must return `true` if startup was successful, and `false` otherwise.
- `bool stop()`
Allows the subclass to clean up prior to termination. The default implementation does nothing but return `true`. The subclass must return `true` if the service has stopped successfully, and `false` otherwise.
- `void syserror(const std::string & msg) const`
- `void error(const std::string & msg) const`
- `void warning(const std::string & msg) const`

- `void trace(const std::string & msg) const`
Convenience functions for logging messages to the communicator's logger.
The `syserror` member function includes a description of the system's current error code.
- `void waitForShutdown()`
Waits indefinitely for the service to shut down. The default implementation invokes `waitForShutdown` on the communicator.

The non-virtual member functions shown in the class definition are described below:

- `bool checkSystem() const`
Returns true if the operating system supports Win32 services or Unix daemons. This function returns false on Windows 95/98/ME.
- `Ice::CommunicatorPtr communicator() const`
Returns the communicator used by the service, as created by `initializeCommunicator`.
- `void configureDaemon(bool chdir, bool close)`
Configures the program to run as a Unix daemon. The `chdir` parameter determines whether the daemon changes its working directory to the root directory. The `close` parameter determines whether the daemon closes unnecessary file descriptors (i.e., `stdin`, `stdout`, etc.).
- `void configureService(const std::string & name)`
Configures the program to run as a Win32 service with the given name.
- `void disableInterrupt()`
Disables the signal handling behavior in `Ice::Service`. When disabled, signals are ignored.
- `void enableInterrupt()`
Enables the signal handling behavior in `Ice::Service`. When enabled, the occurrence of a signal causes the `handleInterrupt` member function to be invoked.
- `int installService(const std::string & name,`
`const std::string & display,`

```
const std::string & executable,
const std::vector<std::string> & args)
```

Registers the program as a Win32 service with the given name. If the `display` parameter is non-empty, it is used as the display name for the service, otherwise the service name is used. If the `executable` parameter is non-empty, it is used as the pathname of the executable, otherwise the pathname of the current executable is used. The values in `args` are passed to the service as command-line arguments at startup. This vector should typically include an option to notify the program that it is being started as a service. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure.

- `static Service * instance()`

Returns the singleton `Ice::Service` instance.

- `int main(int argc, char * argv[])`

The primary entry point of the `Ice::Service` class. The tasks performed by this function are described earlier in this section. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure.

- `std::string name() const`

Returns the name of the service. If the program is running as a Win32 service, the return value is the Win32 service name, otherwise it returns the value of `argv[0]`.

- `int run(int argc, char * argv[])`

Alternative entry point for applications that prefer a different style of service configuration. The program must invoke `configureService` (Win32) or `configureDaemon` (Unix) in order to run as a service. The tasks performed by this function are described earlier in this section. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure.

- `bool service() const`

Returns true if the program is running as a Win32 service or Unix daemon, or false otherwise.

- `int startService(const std::string & name,`

```
const std::vector<std::string> & args)
```

Starts the Win32 service using the Service Control Manager. The values in `args` are passed to the service as command-line options at startup. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure.

- `int stopService(const std::string & name)`
Stops the Win32 service using the Service Control Manager. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure.
- `int uninstallService(const std::string & name)`
Removes a Win32 service using the Service Control Manager. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure.

Unix Daemons

On Unix platforms, `Ice::Service` recognizes the following command-line options:

- **--daemon**

Indicates that the program should run as a daemon. This involves the creation of a background child process in which `Service::main` performs its tasks. The parent process does not terminate until the child process has successfully invoked the `start` member function¹. Unless instructed otherwise, `Ice::Service` changes the current working directory of the child process to the root directory, and closes all unnecessary file descriptors. Note that the file descriptors are not closed until after the communicator is initialized, meaning standard input, standard output, and standard error are available for use during this time. For example, the IceSSL plug-in may need to prompt for a passphrase on standard input, or Ice may print the child's process id on standard output if the property `Ice.PrintProcessId` is set.

- **--noclose**

Prevents `Ice::Service` from closing unnecessary file descriptors. This can be useful during debugging and diagnosis because it provides access to the output from the daemon's standard output and standard error.

- **--nochdir**

Prevents `Ice::Service` from changing the current working directory.

The `--noclose` and `--nochdir` options can only be specified in conjunction with `--daemon`. These options are removed from the argument vector that is passed to the `start` member function.

1. This behavior avoids the uncertainty often associated with starting a daemon from a shell script, because it ensures that the command invocation does not complete until the daemon is ready to receive requests.

Win32 Services

On Win32 platforms², `Ice::Service` starts the application as a Windows service if the `--service` option is specified:

- **--service NAME**

Run as a Windows service named **NAME**. This option is removed from the argument vector that is passed to the `start` member function.

Before an application can run as a Windows service, however, it must be installed, therefore the `Ice::Service` class supports several additional command-line options for performing administrative duties:

- **--install NAME [--display DISP] [--executable EXEC] [ARG ...]**

Installs the service **NAME**. If the `--display` option is specified, use **DISP** as the display name of the service, otherwise use **NAME**. If the `--executable` option is specified, use **EXEC** as the service executable pathname, otherwise use the pathname of the executable used to invoke `--install`. Any additional arguments are passed unchanged to the `Service::start` member function. Note that this command automatically adds the command-line option `--service NAME` to the set of arguments passed to the service at startup, therefore it is not necessary to specify those options explicitly.

- **--uninstall NAME**

Removes the service **NAME**. If the service is currently active, it must be stopped before it can be uninstalled.

- **--start NAME [ARG ...]**

Starts the service **NAME**. Any additional arguments are passed unchanged to the `Service::start` member function.

- **--stop NAME**

Stops the service **NAME**.

An error occurs if more than one administrative command is specified, or if the `--service` option is specified in conjunction with an administrative command. The program terminates immediately after executing the administrative command.

The `Ice::Service` class supports the Windows service control codes `SERVICE_CONTROL_INTERROGATE` and `SERVICE_CONTROL_STOP`. Upon

2. Windows services are not supported on Windows 95/98/ME.

receipt of `SERVICE_CONTROL_STOP`, `Ice::Service` invokes the shutdown member function.

Logging Considerations

When an application is running as a Unix daemon or Windows service, the default implementation of `Ice::Logger` is usually not appropriate because its output is sent to standard error and is therefore lost. The application can implement a custom logger, or it can use one of the alternatives provided by Ice:

- On Unix, the `Ice.UseSyslog` property selects a logger implementation that uses the syslog facility.
- On Windows, the `Ice.UseEventLog` property causes log messages to be recorded in the Windows event log. The logger implementation automatically adds the necessary key to the registry to enable the service to use the event log.

Note that when preparing to start a Windows service, `Ice::Service` creates a temporary instance of the Windows event logger to use until the communicator is successfully initialized and the communicator's configured logger can be used instead. Therefore, even if a failing application is configured to use a different logger implementation, useful diagnostic information may have been recorded in the Windows event log.

If an `Ice::Service` subclass needs to manually install a logger implementation, the subclass should override the `initializeCommunicator` member function.

10.4 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing virtual functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

10.4.1 Skeleton Classes

On the client side, interfaces map to proxy classes (see Section 6.11). On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has a pure virtual member function for each operation on the corresponding interface. For

example, consider the Slice definition for the `Node` interface we defined in Chapter 5 once more:

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };
    // ...
};
```

The Slice compiler generates the following definition for this interface:

```
namespace Filesystem {

    class Node : virtual public Ice::Object {
    public:
        virtual std::string name(const Ice::Current & =
                                Ice::Current()) const = 0;
        // ...
    };
    // ...
}
```

For the moment, we will ignore a number of other member functions of this class. The important points to note are:

- As for the client side, Slice modules are mapped to C++ namespaces with the same name, so the skeleton class definition is nested in the namespace `Filesystem`.
- The name of the skeleton class is the same as the name of the Slice interface (`Node`).
- The skeleton class contains a pure virtual member function for each operation in the Slice interface.
- The skeleton class is an abstract base class because its member functions are pure virtual.
- The skeleton class inherits from `Ice::Object` (which forms the root of the Ice object hierarchy).

10.4.2 Servant Classes

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

```
#include <Filesystem.h> // Slice-generated header

class NodeI : public virtual Filesystem::Node {
public:
    NodeI(const std::string &);
    virtual std::string name(const Ice::Current &) const;
private:
    std::string _name;
};
```

By convention, servant classes have the name of their interface with an I-suffix, so the servant class for the Node interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can chose any name you prefer for your servant classes.)

Note that `NodeI` inherits from `Filesystem::Node`, that is, it derives from its skeleton class. It is a good idea to always use virtual inheritance when defining servant classes. Strictly speaking, virtual inheritance is necessary only for servants that implement interfaces that use multiple inheritance; however, the `virtual` keyword does no harm and, if you add multiple inheritance to an interface hierarchy half-way through development, you do not have to go back and add a `virtual` keyword to all your servant classes.

As far as Ice is concerned, the `NodeI` class must implement only a single member function: the pure virtual `name` function that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. Obviously, the constructor initializes the `_name` member and the `name` function returns its value:

```
NodeI::NodeI(const std::string & name) : _name(name)
{
}

std::string
NodeI::name(const Ice::Current &) const
{
    return _name;
}
```

Normal, idempotent, and nonmutating Operations

The `name` member function of the `NodeI` skeleton on page 252 is a `const` member function. The `const` keyword is added by the Slice compiler because

name is a **nonmutating** operation (see Section 3.8.1). In contrast, normal operations and **idempotent** operations are **non-const** member functions. For example, the following interfaces contains a normal operation, an **idempotent** operation, and a **nonmutating** operation:

```
interface Example {  
    void normalOp();  
    idempotent void idempotentOp();  
    nonmutating void nonmutatingOp();  
};
```

The skeleton class for this interface looks like this:

```
class Example : virtual public Ice::Object {  
public:  
    virtual void normalOp(const Ice::Current &  
                           = Ice::Current()) = 0;  
    virtual void idempotentOp(const Ice::Current &  
                             = Ice::Current()) = 0;  
    virtual void nonmutatingOp(const Ice::Current &  
                               = Ice::Current()) const = 0;  
    // ...  
};
```

Note that only the **nonmutating** operation is mapped as a **const** member function; normal and **idempotent** operations are ordinary member functions.

10.5 Parameter Passing

For each parameter of a Slice operation, the C++ mapping generates a corresponding parameter for the virtual member function in the skeleton. In addition, every operation has an additional, trailing parameter of type `Ice::Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` member function of the `Node` skeleton class has a single parameter of type `Ice::Current`. We explain the purpose of this parameter in Section 16.5 and will ignore it for now.

Parameter passing on the server side follows the rules for the client side:

- in-parameters are passed by value or `const` reference.
- out-parameters are passed by reference.
- return values are passed by value

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

```
interface Example {
    string op(string sin, out string sout);
};
```

The generated skeleton class for this interface looks as follows:

```
class Example : virtual public ::Ice::Object {
public:
    virtual std::string
        op(const std::string &, std::string &,
           const Ice::Current & = Ice::Current()) = 0;
    // ...
};
```

As you can see, there are no surprises here. For example, we could implement `op` as follows:

```
std::string
ExampleI::op(const std::string & sin,
            std::string & sout,
            const Ice::Current &)
{
    cout << sin << endl;          // In parameters are initialized
    sout = "Hello World!";         // Assign out parameter
    return "Done";                 // Return a string
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal C++ rules and do not require special-purpose API calls or memory management.³

3. This is in sharp contrast to the CORBA C++ mapping, which has very complex parameter passing rules that make it all too easy to leak memory or cause undefined behavior.

10.6 Raising Exceptions

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

```
void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    // Try to write the file contents here...
    // Assume we are out of space...
    if (error) {
        Filesystem::GenericError e;
        e.reason = "file too large";
        throw e;
    }
};
```

No memory management issues arise in the presence of exceptions.

Note that the Slice compiler never generates exception specifications for operations, regardless of whether the corresponding Slice operation definition has an exception specification or not. This is deliberate: C++ exception specifications do not add any value and are therefore not used by the Ice C++ mapping. (See [20] for an excellent treatment of the problems associated with C++ exception specifications.)

If you throw an arbitrary C++ exception (such as an `int` or other unexpected type), the Ice run time catches the exception and then returns an `UnknownLocalException` to the client. Similarly, if you throw an “impossible” user exception (a user exception that is not listed in the exception specification of the operation), the client receives an `UnknownUserException`.

The same is true for throwing Ice system exceptions: the client receives an `UnknownLocalException` if you throw, for example, a `MemoryLimitException`.⁴ For that reason, you should never throw system exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

4. There are three system exceptions that are not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`. We discuss these exceptions in more detail in Section 4.8.4 and Chapter 18.

10.7 Object Incarnation

Having created a servant class such as the rudimentary `NodeI` class in Section 10.4.2, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must follow the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

10.7.1 Instantiating a Servant

Instantiating a servant means to allocate an instance on the heap:

```
NodePtr servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance on the heap and assigns its address to a smart pointer of type `NodePtr` (see also page 182). This works because `NodeI` is derived from `Node`, so a smart pointer of type `NodePtr` can also look after an instance of type `NodeI`. However, if we want to invoke a member function of the `NodeI` class at this point, we have a problem: we cannot access member functions of the `NodeI` class through a `NodePtr` smart pointer. (The C++ type rules prevent us from accessing a member of a derived class through a pointer to a base class.) To get around this, we can modify the code as follows:

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI("Fred");
```

This code makes use of the smart pointer template we presented in Section 6.14.5 by defining `NodeIPtr` as a smart pointer to `NodeI` instances. Whether you use a smart pointer of type `NodePtr` or `NodeIPtr` depends solely on whether you want to invoke a member function of the `NodeI` derived class; if not, it is sufficient to use a `NodePtr` and you need not define the `NodeIPtr` type.

Whether you use `NodePtr` or `NodeIPtr`, the advantages of using a smart pointer class should be obvious from the discussion in Section 6.14.5: they make it impossible to accidentally leak memory.

10.7.2 Creating an Identity

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.⁵ An Ice object identity is a structure with the following Slice definition:

```
module Ice {  
    struct Identity {  
        string name;  
        string category;  
    };  
    // ...  
};
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (See Section 16.6 for a discussion of the `category` field.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

```
Ice::Identity id;  
id.name = "Fred"; // Not unique, but good enough for now
```

10.7.3 Activating a Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

```
_adapter->add(servant, id);
```

Note the two arguments to `add`: the smart pointer to the servant and the object identity. Calling `add` on the object adapter adds the servant pointer and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

5. The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity. See XREF for further discussion.

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant pointer from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state (see Section 16.3.5), client requests are dispatched to the servant as soon as you call `add`.

Servant Life Time and Reference Counts

Putting the preceding points together, we can write a simple function that instantiates and activates one of our `NodeI` servants. For this example, we use a simple helper function called `activateServant` that creates and activates a servant with a given identity:

```
void
activateServant(const string & name)
{
    NodePtr servant = new NodeI(name);           // Refcount == 1
    Ice::Identity id;
    id.name = name;
    _adapter->add(servant, id);                // Refcount == 2
}                                               // Refcount == 1
```

Note that we create the servant on the heap and that, once `activateServant` returns, we lose the last remaining handle to the servant (because the `servant` variable goes out of scope). The question is, what happens to the heap-allocated servant instance? The answer lies in the smart pointer semantics:

- When the new servant is instantiated, its reference count is initialized to 0.
- Assigning the servant's address to the `servant` smart pointer increments the servant's reference count to 1.
- Calling `add` passes the `servant` smart pointer to the object adapter which keeps a copy of the handle internally. This increments the reference count of the servant to 2.
- When `activateServant` returns, the destructor of the `servant` variable decrements the reference count of the servant to 1.

The net effect is that the servant is retained on the heap with a reference count of 1 for as long as the servant is in the servant map of its object adapter. (If we deactivate the servant, that is, remove it from the servant map, the reference count drops to zero and the memory occupied by the servant is reclaimed; we discuss these life cycle issues in XREF.)

10.7.4 UUIDs as Identities

As we discussed in Section 2.5.1, the Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) [14] as identities. The `IceUtil` namespace contains a helper function to create such identities:

```
#include <IceUtil/UUID.h>
#include <iostream>

int
main()
{
    cout << IceUtil::generateUUID() << endl;
}
```

When executed, this program prints a unique string such as 5029a22c-e333-4f87-86b1-cd5e0fcce509. Each call to `generateUUID` creates a string that differs from all previous ones.⁶ You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can rewrite the code on page 258 like this:

```
void
activateServant(const string & name)
{
    NodePtr servant = new NodeI(name);
    _adapter->addWithUUID(servant);
}
```

6. Well, almost: eventually, the UUID algorithm wraps around and produces strings that repeat themselves, but this will not happen until approximately the year 3400.

10.7.5 Creating Proxies

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Chapter 3. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI(name);
NodePrx proxy = NodePrx::uncheckedCast(
    _adapter->addWithUUID(servant));

// Pass proxy to client...
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice::ObjectPrx`.

Direct Proxy Creation

The object adapter offers an operation to create a proxy for a given identity:

```
module Ice {
    local interface ObjectAdapter {
        Object* createProxy(Identity id);
        // ...
    };
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

```
Ice::Identity id;
id.name = IceUtil::generateUUID();
ObjectPrx o = _adapter->createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in XREF.)

10.8 Summary

This chapter presented the server-side C++ mapping. Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanism to the client side: a small API to initialize and finalize the run time, plus a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Even though the examples in this chapter are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers (which we discuss in Chapter 16), you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any C++ mapping rules beyond those we described here.

Chapter 11

Developing a File System Server in C++

11.1 Chapter Overview

In this chapter, we present the source code for a C++ server that implements the file system we developed in Chapter 5 (see Chapter 13 for the Java version of the server). The code we present here is fully functional, apart from the required interlocking for threads. (We examine threading issues in detail in Chapter 15.)

11.2 Implementing a File System Server

We have now seen enough of the server-side C++ mapping to implement a server for the file system we developed in Chapter 5. (You may find it useful to review the Slice definition for our file system in Section 5 before studying the source code.)

Our server is composed of two source files:

- `Server.cpp`

This file contains the server main program.

- `FilesystemI.cpp`

This file contains the implementation for the file system servants.

11.2.1 The Server main Program

Our server main program, in the file `Server.cpp`, uses the `Ice::Application` class we discussed in Section 10.3.1. The `run` method installs a signal handler, creates an object adapter, instantiates a few servants for the directories and files in the file system, and then activates the adapter. This leads to a main program as follows:

```
#include <FilesystemI.h>
#include <Ice/Application.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : virtual public Ice::Application {
public:
    virtual int run(int, char * []) {
        // Create an object adapter (stored in the NodeI::_adapter
        // static member)
        //
        NodeI::_adapter =
            communicator()->createObjectAdapterWithEndpoints(
                "SimpleFilesystem", "default -p 10000");

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI("/", 0);

        // Create a file called "README" in the root directory
        //
        FilePtr file = new FileI("README", root);
        Lines text;
        text.push_back("This file system contains"
                      "a collection of poetry.");
        file->write(text);

        // Create a directory called "Coleridge" in
        // the root directory
        //
        DirectoryIPtr coleridge
            = new DirectoryI("Coleridge", root);

        // Create a file called "Kubla_Khan" in the
        // Coleridge directory
        //
        file = new FileI("Kubla_Khan", coleridge);
```

```
    text.erase(text.begin(), text.end());
    text.push_back("In Xanadu did Kubla Khan");
    text.push_back("A stately pleasure-dome decree:");
    text.push_back("Where Alph, the sacred river, ran");
    text.push_back("Through caverns measureless to man");
    text.push_back("Down to a sunless sea.");
    file->write(text);

    // All objects are created, allow client requests now
    //
    NodeI::_adapter->activate();

    // Wait until we are done
    //
    communicator()->waitForShutdown();
    if (interrupted()) {
        cerr << appName()
            << ": received signal, shutting down" << endl;
    }
    return 0;
};

int
main(int argc, char* argv[])
{
    FilesystemApp app;
    return app.main(argc, argv);
}
```

There is quite a bit of code here, so let us examine each section in detail:

```
#include <FilesystemI.h>
#include <Ice/Application.h>

using namespace std;
using namespace Filesystem;
```

The code includes the header file `FilesystemI.h` (see page 274). That file includes `Ice/Ice.h` as well as the header file that is generated by the Slice compiler, `Filesystem.h`. Because we are using `Ice::Application`, we need to include `Ice/Application.h` as well.

Two using declarations, for the namespaces `std` and `Filesystem`, permit us to be a little less verbose in the source code.

The next part of the source code is the definition of `FilesystemApp`, which derives from `Ice::Application` and contains the main application logic in its `run` method:

```
class FilesystemApp : virtual public Ice::Application {
public:
    virtual int run(int, char * []) {
        // Create an object adapter (stored in the NodeI::_adapter
        // static member)
        //
        NodeI::_adapter =
            communicator()->createObjectAdapterWithEndpoints(
                "SimpleFilesystem", "default -p 10000");

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI("/", 0);

        // Create a file called "README" in the root directory
        //
        FilePtr file = new FileI("README", root);
        Lines text;
        text.push_back("This file system contains"
                      "a collection of poetry.");
        file->write(text);

        // Create a directory called "Coleridge" in
        // the root directory
        //
        DirectoryIPtr coleridge
            = new DirectoryI("Coleridge", root);

        // Create a file called "Kubla_Khan" in the
        // Coleridge directory
        //
        file = new FileI("Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);

        // All objects are created, allow client requests now
        //
```

```

        NodeI::_adapter->activate();

        // Wait until we are done
        //
        communicator()->waitForShutdown();
        if (interrupted()) {
            cerr << appName()
                << ": received signal, shutting down" << endl;
        }
        return 0;
    };
}

```

Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown in Figure 11.1.

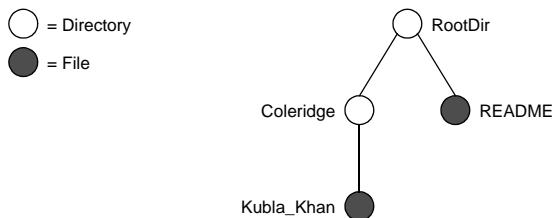


Figure 11.1. A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a handle to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent handle.) Thus, the statement

```
DirectoryIPtr root = new DirectoryI("/", 0);
```

creates the root directory, with the name "/" and no parent directory. Note that we use the smart pointer class we discussed in Section 6.14.5 to hold the return value from `new`; that way, we avoid any memory management issues. The type `DirectoryIPtr` is defined as follows in a header file `FilesystemI.h` (see page 274):

```
typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;
```

Here is the code that establishes the structure in Figure 11.1:

```
// Create the root directory (with name "/" and no parent)
//
DirectoryIPtr root = new DirectoryI("/", 0);

// Create a file called "README" in the root directory
//
FilePtr file = new FileI("README", root);
Lines text;
text.push_back("This file system contains"
    "a collection of poetry.");
file->write(text);

// Create a directory called "Coleridge" in
// the root directory
//
DirectoryIPtr coleridge
    = new DirectoryI("Coleridge", root);

// Create a file called "Kubla_Khan" in the
// Coleridge directory
//
file = new FileI("Kubla_Khan", coleridge);
text.erase(text.begin(), text.end());
text.push_back("In Xanadu did Kubla Khan");
text.push_back("A stately pleasure-dome decree:");
text.push_back("Where Alph, the sacred river, ran");
text.push_back("Through caverns measureless to man");
text.push_back("Down to a sunless sea.");
file->write(text);
```

We first create the root directory and a file README within the root directory. (Note that we pass the handle to the root directory as the parent pointer when we create the new node of type FileI.)

The next step is to fill the file with text:

```
FilePtr file = new FileI("README", root);
Lines text;
text.push_back("This file system contains"
    "a collection of poetry.");
file->write(text);
```

Recall from Section 6.7.3 that Slice sequences map to STL vectors. The Slice type `Lines` is a sequence of strings, so the C++ type `Lines` is a vector of strings; we add a line of text to our `README` file by calling `push_back` on that vector.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

```
file->write(text);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a smart class pointer (of type `FilePtr`) and *not* via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place—such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary C++ function call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in Figure 11.1.

11.2.2 The Servant Class Definitions

We must provide servants for the concrete interfaces in our Slice specification, that is, we must provide servants for the `File` and `Directory` interfaces in the C++ classes `FileI` and `DirectoryI`. This means that our servant classes might look as follows:

```
namespace Filesystem {
    class FileI : virtual public File {
        // ...
    };

    class DirectoryI : virtual public Directory {
        // ...
    };
}
```

This leads to the C++ class structure as shown in Figure 11.2.

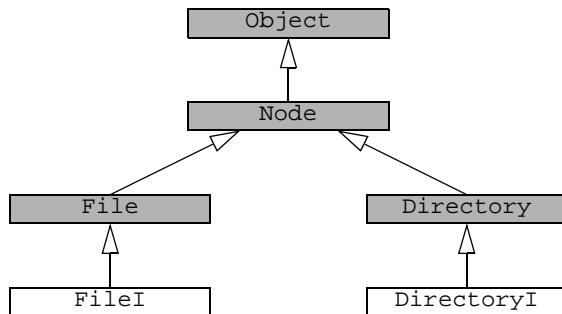


Figure 11.2. File system servants using interface inheritance.

The shaded classes in Figure 11.2 are skeleton classes and the unshaded classes are our servant implementations. If we implement our servants like this, `FileI` must implement the pure virtual operations it inherits from the `File` skeleton (`read` and `write`), as well as the operation it inherits from the `Node` skeleton (`name`). Similarly, `DirectoryI` must implement the pure virtual function it inherits from the `Directory` skeleton (`list`), as well as the operation it inherits from the `Node` skeleton (`name`). Implementing the servants in this way uses interface inheritance from `Node` because no implementation code is inherited from that class.

Alternatively, we can implement our servants using the following definitions:

```

namespace Filesystem {
    class NodeI : virtual public Node {
        // ...
    };

    class FileI : virtual public File,
                  virtual public NodeI {
        // ...
    };

    class DirectoryI : virtual public Directory,
                      virtual public NodeI {
        // ...
    };
}
  
```

This leads to the C++ class structure shown in Figure 11.3.

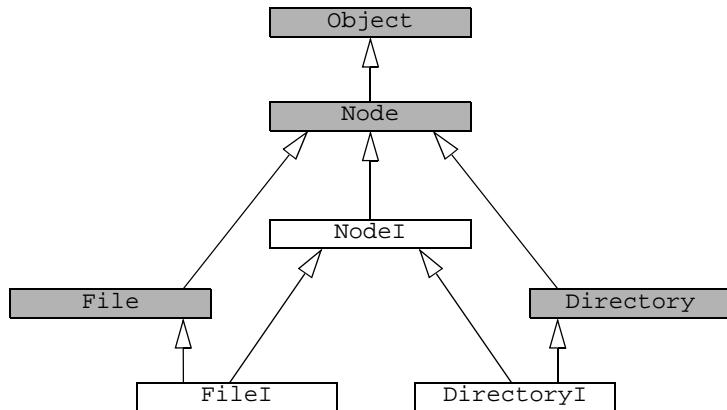


Figure 11.3. File system servants using implementation inheritance.

In this implementation, `NodeI` is a concrete base class that implements the name operation it inherits from the `Node` skeleton. `FileI` and `DirectoryI` use multiple inheritance from `NodeI` and their respective skeletons, that is, `FileI` and `DirectoryI` use implementation inheritance from their `NodeI` base class.

Either implementation approach is equally valid. Which one to choose simply depends on whether we want to re-use common code provided by `NodeI`. For the implementation that follows, we have chosen the second approach, using implementation inheritance.

Given the structure in Figure 11.3 and the operations we have defined in the Slice definition for our file system, we can add these operations to the class definition for our servants:

```

namespace Filesystem {
    class NodeI : virtual public Node {
    public:
        virtual std::string name(const Ice::Current &) const;
    };

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        virtual Filesystem::Lines read(const Ice::Current &) const;
        virtual void write(const Filesystem::Lines &,
                          const Ice::Current &);
    };
}
  
```

```

};

class DirectoryI : virtual public Directory,
                   virtual public Filesystem::NodeI {
public:
    virtual Filesystem::NodeSeq list(const Ice::Current &) const;
};

}

```

This simply adds signatures for the operation implementations to each class. (Note that the signatures must exactly match the operation signatures in the generated skeleton classes—if they do not match exactly, you end up overloading the pure virtual function in the base class instead of overriding it, meaning that the servant class cannot be instantiated because it will still be abstract. To avoid signature mismatches, you can copy the signatures from the generated header file (`Filesystem.h`).)

Now that we have the basic structure in place, we need to think about other methods and data members we need to support our servant implementation. Typically, each servant class hides the copy constructor and assignment operator, and has a constructor to provide initial state for its data members. Given that all nodes in our file system have both a name and a parent directory, this suggests that the `NodeI` class should implement the functionality relating to tracking the name of each node, as well as the parent–child relationships:

```

namespace Filesystem {
    class DirectoryI;
    typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

    class NodeI : virtual public Node {
public:
    virtual std::string name(const Ice::Current &) const;
    NodeI(const std::string &, const DirectoryIPtr & parent);
    static Ice::ObjectAdapterPtr _adapter;
private:
    const std::string _name;
    DirectoryIPtr _parent;
    NodeI(const NodeI &);                                // Copy forbidden
    void operator=(const NodeI &);                         // Assignment forbidden
};
}

```

The `NodeI` class has a private data member to store its name (of type `std::string`) and its parent directory (of type `DirectoryIPtr`). The constructor accepts parameters that set the value of these data members. For the

root directory, by convention, we pass a null handle to the constructor to indicate that the root directory has no parent. We have also added a public static variable to hold a smart pointer to the (single) object adapter we use in our server; that variable is initialized by the `Filesystem::run` method on page 266.

The `FileI` servant class must store the contents of its file, so it requires a data member for this. We can conveniently use the generated `Lines` type (which is a `std::vector<std::string>`) to hold the file contents, one string for each line. Because `FileI` inherits from `NodeI`, it also requires a constructor that accepts the file name and the parent directory, leading to the following class definition:

```
namespace Filesystem {
    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
public:
    virtual Filesystem::Lines read(const Ice::Current &) const;
    virtual void write(const Filesystem::Lines &,
                      const Ice::Current &);
    FileI(const std::string &, const DirectoryIPtr &);
private:
    Lines _lines;
};
}
```

For directories, each directory must store its list of child nodes. We can conveniently use the generated `NodeSeq` type (which is a `vector<NodePrx>`) to do this. Because `DirectoryI` inherits from `NodeI`, we need to add a constructor to initialize the directory name and its parent directory. As we will see shortly, we also need a private helper function, `addChild`, to make it easier to connect a newly created directory to its parent. This leads to the following class definition:

```
namespace Filesystem {
    class DirectoryI : virtual public Directory,
                      virtual public Filesystem::NodeI {
public:
    virtual Filesystem::NodeSeq list(const Ice::Current &) const;
    DirectoryI(const std::string &, const DirectoryIPtr &);
    void addChild(NodePrx child);
private:
    NodeSeq _contents;
};
}
```

Putting all this together, we end up with a servant header file, `FilesystemI.h`, as follows:

```
#include <Ice/Ice.h>
#include <Filesystem.h>

namespace Filesystem {
    class DirectoryI;
    typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

    class NodeI : virtual public Node {
public:
    virtual std::string name(const Ice::Current &) const;
    NodeI(const std::string &, const DirectoryIPtr & parent);
    static Ice::ObjectAdapterPtr _adapter;
private:
    const std::string _name;
    DirectoryIPtr _parent;
    NodeI(const NodeI &); // Copy forbidden
    void operator=(const NodeI &); // Assignment forbidden
};

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
public:
    virtual Filesystem::Lines read(const Ice::Current &) const;
    virtual void write(const Filesystem::Lines &,
                      const Ice::Current &);

    FileI(const std::string &, const DirectoryIPtr &);

private:
    Lines _lines;
};

    class DirectoryI : virtual public Directory,
                      virtual public Filesystem::NodeI {
public:
    virtual Filesystem::NodeSeq list(const Ice::Current &) const;
    DirectoryI(const std::string &, const DirectoryIPtr &);

    void addChild(NodePrx child);

private:
    NodeSeq _contents;
};
}
```

11.2.3 The Servant Implementation

The implementation of our servants is mostly trivial, following from the class definitions in our `FilesystemI.h` header file.

Implementing `FileI`

The implementation of the `read` and `write` operations for files is trivial: we simply store the passed file contents in the `_lines` data member. The constructor is equally trivial, simply passing its arguments through to the `NodeI` base class constructor:

```
Filesystem::Lines  
Filesystem::FileI::read(const Ice::Current &) const  
{  
    return _lines;  
}  
  
void  
Filesystem::FileI::write(const Filesystem::Lines & text,  
                        const Ice::Current &)  
{  
    _lines = text;  
}  
  
Filesystem::FileI::FileI(const string & name,  
                        const DirectoryIPtr & parent  
) : NodeI(name, parent)  
{  
}
```

Implementing `DirectoryI`

The implementation of `DirectoryI` is equally trivial: the `list` operation simply returns the `_contents` data member and the constructor passes its arguments through to the `NodeI` base class constructor:

```
Filesystem::NodeSeq  
Filesystem::DirectoryI::list(const Ice::Current &) const  
{  
    return _contents;  
}  
  
Filesystem::DirectoryI::DirectoryI(const string & name,  
                                  const DirectoryIPtr & parent  
) : NodeI(name, parent)
```

```
{
}

void
Filesystem::DirectoryI::addChild(const NodePrx child)
{
    _contents.push_back(child);
}
```

The only noteworthy thing is the implementation of `addChild`: when a new directory or file is created, the constructor of the `NodeI` base class calls `addChild` on its own parent, passing it the proxy to the newly-created child. The implementation of `addChild` appends the passed reference to the contents list of the directory it is invoked on (which is the parent directory).

Implementing `NodeI`

The name operation of our `NodeI` class is again trivial: it simply returns the `_name` data member:

```
std::string
Filesystem::NodeI::name(const Ice::Current &) const
{
    return _name;
}
```

Most of the meat of our implementation is in the `NodeI` constructor. It is here that we create the proxy for each node and connect parent and child nodes:

```
Filesystem::NodeI::NodeI(const string & name,
                       const DirectoryIPtr & parent
                      ) : _name(name), _parent(parent)
{
    // Create an identity. The parent has the fixed identity "/"
    //
    Ice::Identity myID = Ice::stringToIdentity(parent
                                                ? IceUtil::generateUUID()
                                                : "RootDir");

    // Create a proxy for the new node and add it as
    // a child to the parent
    //
    NodePrx thisNode
        = NodePrx::uncheckedCast(_adapter->createProxy(myID));
    if (parent)
        parent->addChild(thisNode);
```

```
// Activate the servant  
//  
_adapter->add(this, myID);  
}
```

The first step is to create a unique identity for each node. For the root directory, we use the fixed identity "RootDir". This allows the client to create a proxy for the root directory (see Section 7.2). For directories other than the root directory, we use a UUID as the identity (see page 259).

The next step is to create a proxy for the child and to add the child to the parent directory's content list by calling `addChild`. This connects the child to the parent.

Finally, we need to activate the servant so the Ice run time knows about the servant's existence, so we call `add` on the object adapter.

This completes our servant implementation. The complete source code is shown here once more:

```
#include <FilesystemI.h>
#include <IceUtil/UUID.h>
#include <time.h>
```

```
// Create a proxy for the new node and add it
// as a child to the parent
//
NodePrx thisNode
    = NodePrx::uncheckedCast(_adapter->createProxy(myID));
if (parent)
    parent->addChild(thisNode);

// Activate the servant
//
_adapter->add(this, myID);
}

// Slice File::read() operation

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    return _lines;
}

// Slice File::write() operation

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                       const Ice::Current &)
{
    _lines = text;
}

// FileI constructor

Filesystem::FileI::FileI(const string & name,
                       const DirectoryIPtr & parent
                      ) : NodeI(name, parent)
{
}

// Slice Directory::list() operation

Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current &) const
{
    return _contents;
}
```

```
// DirectoryI constructor

Filesystem::DirectoryI::DirectoryI(const string & name,
                                    const DirectoryIPtr & parent
                                   ) : NodeI(name, parent)
{
}

// addChild is called by the child in order to add
// itself to the _contents member of the parent

void
Filesystem::DirectoryI::addChild(const NodePrx child)
{
    _contents.push_back(child);
}
```

11.3 Summary

This chapter showed how to implement a complete server for the file system we defined in Chapter 5. Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

Note that the server code we presented here is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the `_lines` data member while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, it is trivial to make the `read` and `write` operations thread-safe: a single data member and two lines of source code are sufficient to achieve this. We discuss how to write thread-safe servant implementations in Chapter 15.

Chapter 12

Server-Side Slice-to-Java Mapping

12.1 Chapter Overview

In this chapter, we present the server-side Slice-to-Java mapping (see Chapter 10 for the server-side Slice-to-C++ mapping). Section 12.3 discusses how to initialize and finalize the server-side run time, sections 12.4 to 12.7 show how to implement interfaces and operations, and Section 12.8 discusses how to register objects with the server-side Ice run time.

12.2 Introduction

The mapping for Slice data types to Java is identical on the client side and server side. This means that everything in Chapter 8 also applies to the server side. However, for the server side, there are a few additional things you need to know, specifically:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions
- how to create servants and register them with the Ice run time.

We discuss these topics in the remainder of this chapter.

12.3 The Server-Side main Function

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. As for the client side, you must initialize the Ice run time by calling `Ice.Util.initialize` before you can do anything else in your server. `Ice.Util.initialize` returns a reference to an instance of an `Ice.Communicator`:

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // ...
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        }
        // ...
    }
}
```

`Ice.Util.initialize` accepts the argument vector that is passed to `main` by the operating system. The function scans the argument vector for any command-line options that are relevant to the Ice run time, but does not remove those options.¹ If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your `main` function, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` waits for any operation invocations that may still be running to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your `main` function to terminate without calling `destroy` first; doing so has undefined behavior.

1. The semantics of Java arrays prevents `Ice.Util.initialize` from modifying the size of the argument vector. However, another overloading of `Ice.Util.initialize` is provided that allows the application to obtain a new argument vector with the Ice options removed.

The general shape of our server-side main function is therefore as follows:

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // ...
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        }
        if (ic != null)
            ic.destroy();
        System.exit(status);
    }
}
```

Note that the code places the call to `Ice::initialize` into a `try` block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

12.3.1 The `Ice.Application` Class

The preceding structure for the main function is so common that Ice offers a class, `Ice.Application`, that encapsulates all the correct initialization and finalization activities. The synopsis of the class is as follows (with some detail omitted for now):

```
package Ice;

public abstract class Application {
    public Application()

    public final int main(String appName, String[] args)

    public final int
        main(String appName, String[] args, String configFile)

    public abstract int run(String[] args);
```

```
    public static String appName()

    public static Communicator communicator()

    // ...
}
```

The intent of this class is that you specialize `Ice.Application` and implement the abstract `run` method in your derived class. Whatever code you would normally place in `main` goes into the `run` method instead. Using `Ice.Application`, our program looks as follows:

```
public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Server code here...

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

The `Application.main` function does the following:

1. It installs an exception handler for `java.lang.Exception`. If your code fails to handle an exception, `Application.main` prints the name of an exception and a stack trace on `System.err` before returning with a non-zero return value.
2. It initializes (by calling `Ice.Util.initialize`) and finalizes (by calling `Communicator.destroy`) a communicator. You can get access to the communicator for your server by calling the static `communicator` accessor.
3. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.

4. It provides the name of your application via the static `appName` member function. The return value from this call is the first argument in the call to `Application.main`, so you can get at this name from anywhere in your code by calling `Ice.Application.appName` (which is usually required for error messages). In the example above, the return value from `appName` would be `Server`.
5. It installs a shutdown hook that properly shuts down the communicator.

Using `Ice.Application` ensures that your program properly finalizes the Ice run time, whether your server terminates normally or in response to an exception. We recommend that all your programs use this class; doing so makes your life easier. In addition `Ice.Application` also provides features for signal handling and configuration that you do not have implement yourself when you use this class.

Using `Ice.Application` on the Client Side

You can use `Ice.Application` for your clients as well: simply implement a class that derives from `Ice.Application` and place the client code into its `run` method. The advantage of this approach is the same as for the server side: `Ice.Application` ensures that the communicator is destroyed correctly even in the presence of exceptions.

Catching Signals

The simple server we developed in Chapter 3 had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

Java does not provide direct support for signals, but it does allow an application to register a *shutdown hook* that is invoked when the JVM is shutting down. There are several events which trigger JVM shutdown, such as a call to `System.exit` or an interrupt signal from the operating system, but the shutdown hook is not provided with the reason for the shut down.

`Ice.Application` registers a shutdown hook by default, allowing you to cleanly terminate your application prior to JVM shutdown:

```
package Ice;

public abstract class Application {
    // ...

    synchronized public static void shutdownOnInterrupt()

    synchronized public static void defaultInterrupt()

    synchronized public static boolean interrupted()
}
```

The member functions behave as follows:

- `shutdownOnInterrupt`

This function installs a shutdown hook that cleanly shuts down your application by calling `shutdown` on the communicator. This is the default behavior.

- `defaultInterrupt`

This function removes the shutdown hook.

- `interrupted`

This function returns true if the shutdown hook caused the communicator to shut down, false otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by the JVM. This is useful, for example, for logging purposes.

By default, `Ice.Application` behaves as if `shutdownOnInterrupt` was invoked, therefore our server `main` function requires no change to ensure that the program terminates cleanly on JVM shutdown. However, we add a diagnostic to report the occurrence, so our `main` function now looks like:

```
public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Server code here...

        if (interrupted())
            System.err.println(appName() + ": terminating");

        return 0;
    }

    public static void
    main(String[] args)
```

```
{  
    Server app = new Server();  
    int status = app.main("Server", args);  
    System.exit(status);  
}  
}
```

Ice.Application and Properties

Apart from the functionality shown in this section, `Ice.Application` also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or port number for a server. The `main` function of `Ice.Application` is overloaded; the second version allows you to specify the name of a configuration file that will be processed during initialization. We discuss Ice properties in more detail in Chapter 14.

Limitations of `Ice.Application`

`Ice.Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use `Ice.Application`. Instead, you must structure your code as we saw in Chapter 3 (taking care to always destroy the communicator).

12.4 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing member functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

12.4.1 Skeleton Classes

On the client side, interfaces map to proxy classes (see Section 5.12). On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has a pure virtual member function for each operation on the corresponding interface. For example, consider the Slice definition for the `Node` interface we defined in Chapter 5 once more:

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };
    // ...
};
```

The Slice compiler generates the following definition for this interface:

```
package Filesystem;

public interface _NodeOperations
{
    String name(Ice.Current current);
}

public interface Node extends Ice.Object, _NodeOperations {}

public abstract class _NodeDisp extends Ice.ObjectImpl
    implements Node
{
    // Mapping-internal code here...
}
```

The important points to note here are:

- As for the client side, Slice modules are mapped to Java packages with the same name, so the skeleton class definitions are part of the `Filesystem` package.
- For each Slice interface `<interface-name>`, the compiler generates a Java interface `_<interface-name>Operations` (`_NodeOperations` in this example). This interface contains a member function for each operation in the Slice interface.
- For each Slice interface `<interface-name>`, the compiler generates a Java interface `<interface-name>` (`Node` in this example). That interface extends both `Ice.Object` and `_<interface-name>Operations`.
- For each Slice interface `<interface-name>`, the compiler generates an abstract class `_<interface-name>Disp` (`_NodeDisp` in this example). This abstract class is the actual skeleton class; it is the base class from which you derive your servant class.

12.4.2 Servant Classes

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the Node interface, you could write:

```
package Filesystem;

public final class NodeI extends _NodeDisp {

    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the Node interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can chose any name you prefer for your servant classes.) Note that `NodeI` extends `_NodeDisp`, that is, it derives from its skeleton class.

As far as Ice is concerned, the `NodeI` class must implement only a single method: the `name` method that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. (Obviously, the constructor initializes the `_name` member and the `name` function returns its value.)

Normal, idempotent, and nonmutating Operations

Whether an operation is an ordinary operation, or an `idempotent` or `nonmutating` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

```
interface Example {
    void normalOp();
    idempotent void idempotentOp();
    nonmutating void nonmutatingOp();
};
```

The operations class for this interface looks like this:

```
public interface _ExampleOperations
{
    void normalOp(Ice.Current current);
    void idempotentOp(Ice.Current current);
    void nonmutatingOp(Ice.Current current);
}
```

Note that the signatures of the member functions are unaffected by the `idempotent` and `nonmutating` qualifiers. (In C++, the `nonmutating` qualifier generates a C++ `const` member function.)

12.5 Parameter Passing

For each parameter of a Slice operation, the Java mapping generates a corresponding parameter for the corresponding method in the `_<interface-name>Operations` interface. In addition, every operation has an additional, trailing parameter of type `Ice.Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` member function of the `_NodeOperations` interface has a single parameter of type `Ice.Current`. We explain the purpose of this parameter in Section 16.5 and will ignore it for now.

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

```
interface Example {
    string op(string sin, out string sout);
};
```

The generated skeleton class for this interface looks as follows:

```
public interface _ExampleOperations
{
    String op(String sin, Ice.StringHolder sout,
              Ice.Current current);
}
```

As you can see, there are no surprises here. For example, we could implement `op` as follows:

```
public final class ExampleI extends _ExampleDisp {  
  
    public String op(String sin, Ice.StringHolder sout,  
                    Ice.Current current)  
    {  
        System.out.println(sin);      // In params are initialized  
        sout.value = "Hello World!"; // Assign out param  
        return "Done";  
    }  
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal Java rules and do not require special-purpose API calls.

12.6 Raising Exceptions

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

```
// ...  
  
public void  
write(String[] text, Ice.Current current)  
throws GenericError  
  
{  
    // Try to write file contents here...  
    // Assume we are out of space...  
    if (error) {  
        GenericError e = new GenericError();  
        e.reason = "file too large";  
        throw e;  
    }  
}
```

If you throw an arbitrary Java run-time exception, the Ice run time catches the exception and then returns an `UnknownLocalException` to the client. The same is true for throwing Ice system exceptions: the client receives an `UnknownLocalException` if you throw, for example, a `MemoryLimitException`.² For that reason, you should never throw system exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

12.7 Tie Classes

The mapping to skeleton classes we saw in Section 12.4 requires the servant class to inherit from its skeleton class. Occasionally, this creates a problem: some class libraries require you to inherit from a base class in order to access functionality provided by the library; because Java does not support multiple inheritance, this means that you cannot use such a class library to implement your servants because your servants cannot inherit from both the library class and the skeleton class simultaneously.

To allow you to still use such class libraries, Ice provides a way to write servants that replaces inheritance with delegation. This approach is supported by *tie classes*. The idea is that, instead of inheriting from the skeleton class, you simply create a class (known as an *implementation class* or *delegate class*) that contains methods corresponding to the operations of an interface. You also use the `--tie` option with the `slice2java` compiler to create a tie class. For example, for the `Node` interface we saw in Section 12.4.1, the `--tie` option causes the compiler to create exactly the same code as we saw previously, but to also emit an additional tie class. For an interface `<interface-name>`, the generated tie class has the name `_<interface-name>Tie`:

```
package Filesystem;

public class _NodeTie extends _NodeDisp {
    public _NodeTie() {}
}
```

2. There are three system exceptions that are not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`. We discuss these exceptions in more detail in Chapter 18.

```
public
_NodeTie(_NodeOperations delegate)
{
    _ice_delegate = delegate;
}

public _NodeOperations
ice_delegate()
{
    return _ice_delegate;
}

public void
ice_delegate(_NodeOperations delegate)
{
    _ice_delegate = delegate;
}

public boolean
equals(java.lang.Object rhs)
{
    if(this == rhs)
    {
        return true;
    }
    if(!(rhs instanceof _NodeTie))
    {
        return false;
    }

    return _ice_delegate.equals(((NodeTie)rhs)._ice_delegate)
;
}

public int
hashCode()
{
    return _ice_delegate.hashCode();
}

public String
name(Ice.Current current)
{
    return _ice_delegate.name(current);
```

```

        }

    private _NodeOperations _ice_delegate;
}

```

This looks a lot worse than it is: in essence, the generated tie class is simply a servant class (it extends `_NodeDisp`) that delegates each invocation of a method that corresponds to a Slice operation to your implementation class (see Figure 12.1).

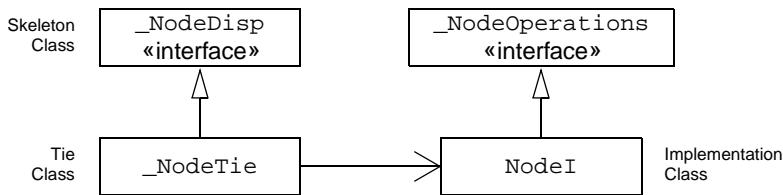


Figure 12.1. A skeleton class, tie class, and implementation class.

Given this machinery, we can create an implementation class for our `Node` interface as follows:

```

package Filesystem;

public final class NodeI implements _NodeOperations {
{
    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}

```

Note that this class is identical to our previous implementation, except that it implements the `_NodeOperations` interface and does not extend `_NodeDisp` (which means that you are now free to extend any other class to support your implementation).

To create a servant, you instantiate your implementation class and the tie class, passing a reference to the implementation instance to the tie constructor:

```
NodeI fred = new NodeI("Fred");           // Create implementation
_NodeTie servant = new _NodeTie(fred);    // Create tie
```

Alternatively, you can also default-construct the tie class and later set its delegate instance by calling `ice_delegate`:

```
_NodeTie servant = new _NodeTie();         // Create tie
// ...
NodeI fred = new NodeI("Fred");           // Create implementation
// ...
servant.ice_delegate(fred);              // Set delegate
```

When using tie classes, it is important to remember that the tie instance is the servant, not your delegate. Furthermore, you must not use a tie instance to incarnate (see Section 12.8) an Ice object until the tie has a delegate. Once you have set the delegate, you must not change it for the lifetime of the tie; otherwise, undefined behavior results.

You should use the tie approach only if you need to, that is, if you need to extend some base class in order to implement your servants: using the tie approach is more costly in terms of memory because each Ice object is incarnated by two Java objects instead of one, the tie and the delegate. In addition, call dispatch for ties is marginally slower than for ordinary servants because the tie forwards each operation to the delegate, that is, each operation invocation requires two function calls instead of one.

Also note that, unless you arrange for it, there is no way to get from the delegate back to the tie. If you need to navigate back to the tie from the delegate, you can store a reference to the tie in a member of the delegate. (The reference can, for example, be initialized by the constructor of the delegate.)

12.8 Object Incarnation

Having created a servant class such as the rudimentary `NodeI` class in Section 12.4.2, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.

2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

12.8.1 Instantiating a Servant

Instantiating a servant means to allocate an instance:

```
Node servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance and assigns its address to a reference of type `Node`. This works because `NodeI` is derived from `Node`, so a `Node` reference can refer to an instance of type `NodeI`. However, if we want to invoke a member function of the `NodeI` class at this point, we must use a `NodeI` reference:

```
NodeI servant = new NodeI("Fred");
```

Whether you use a `Node` or a `NodeI` reference depends purely on whether you want to invoke a member function of the `NodeI` class: if not, a `Node` reference works just as well as a `NodeI` reference.

12.8.2 Creating an Identity

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.³ An Ice object identity is a structure with the following Slice definition:

```
module Ice {  
    struct Identity {  
        string name;  
        string category;  
    };  
    // ...  
};
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the

3. The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity. See XREF for further discussion.

empty string and simply use the `name` field. (See Section 16.5 for a discussion of the `category` field.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

```
Ice.Identity id = new Ice.Identity();
id.name = "Fred"; // Not unique, but good enough for now
```

12.8.3 Activating a Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

```
_adapter.add(servant, id);
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state (see Section 16.3), client requests are dispatched to the servant as soon as you call `add`.

12.8.4 UIDs as Identities

As we discussed in Section 2.5.1, the Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) [14] as identities. The `Ice.Util` package contains a helper function to create such identities:

```
public class Example {  
    public static void  
    main(String[] args)  
    {  
        System.out.println(Ice.Util.generateUUID());  
    }  
}
```

When executed, this program prints a unique string such as 5029a22c-e333-4f87-86b1-cd5e0fcce509. Each call to generateUUID creates a string that differs from all previous ones.⁴ You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation addWithUUID that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

```
_adapter.addWithUUID(new NodeI("Fred"));
```

12.8.5 Creating Proxies

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Chapter 3. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation

The add and addWithUUID servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

4. Well, almost: eventually, the UUID algorithm wraps around and produces strings that repeat themselves, but this will not happen until approximately the year 3400.

```
NodePrx proxy = NodePrxHelper.uncheckedCast(  
    _adapter.addWithUUID(new NodeI("Fred")));
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice.ObjectPrx`.

Direct Proxy Creation

The object adapter offers an operation to create a proxy for a given identity:

```
module Ice {  
    local interface ObjectAdapter {  
        Object* createProxy(Identity id);  
        // ...  
    };  
};
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

```
Ice.Identity id = new Ice.Identity();  
id.name = Ice.Util.generateUUID();  
Ice.ObjectPrx o = _adapter.createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in XREF.)

12.9 Summary

This chapter presented the server-side Java mapping. Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanism to the client side: a small API to initialize and finalize the run time, plus a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Even though the examples in this chapter are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated

servers (which we discuss in Chapter 16), you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any Java mapping rules beyond those we described here.

Chapter 13

Developing a File System Server in Java

13.1 Chapter Overview

In this chapter, we present the source code for a Java server that implements the file system we developed in Chapter 5 (see Chapter 11 for the C++ version of the server). The code we present here is fully functional, apart from the required interlocking for threads. (We examine threading issues in detail in 15.)

13.2 Implementing a File System Server

We have now seen enough of the server-side Java mapping to implement a server for the file system we developed in Chapter 5. (You may find it useful to review the Slice definition for our file system in Section 5 before studying the source code.)

Our server is composed of three source files:

- `Server.java`

This file contains the server main program.

- `Filesystem/DirectoryI.java`

This file contains the implementation for the `Directory` servants.

- `Filesystem/FileI.java`

This file contains the implementation for the `File` servants.

13.2.1 The Server main Program

Our server main program, in the file `Server.java`, uses the `Ice.Application` class we discussed in Section 12.3.1. The `run` method installs a shutdown hook, creates an object adapter, instantiates a few servants for the directories and files in the file system, and then activates the adapter. This leads to a main program as follows:

```
import Filesystem.*;

public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Create an object adapter (stored in the _adapter
        // static members)
        //
        Ice.ObjectAdapter adapter
            = communicator().createObjectAdapterWithEndpoints(
                "SimpleFilesystem", "default -p 10000");
        DirectoryI._adapter = adapter;
        FileI._adapter = adapter;

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI root = new DirectoryI("/", null);

        // Create a file "README" in the root directory
        //
        File file = new FileI("README", root);
        String[] text;
        text = new String[] {
            "This file system contains a collection of poetry."
        };
        try {
            file.write(text, null);
        } catch (GenericError e) {
            System.err.println(e.reason);
        }

        // Create a directory "Coleridge" in the root directory
```

```
//  
DirectoryI coleridge  
= new DirectoryI("Coleridge", root);  
  
// Create a file "Kubla_Khan" in the Coleridge directory  
//  
file = new FileI("Kubla_Khan", coleridge);  
text = new String[]{ "In Xanadu did Kubla Khan",  
                     "A stately pleasure-dome decree:",  
                     "Where Alph, the sacred river, ran",  
                     "Through caverns measureless to man",  
                     "Down to a sunless sea." };  
try {  
    file.write(text, null);  
} catch (GenericError e) {  
    System.err.println(e.reason);  
}  
  
// All objects are created, allow client requests now  
//  
adapter.activate();  
  
// Wait until we are done  
//  
communicator().waitForShutdown();  
  
return 0;  
}  
  
public static void  
main(String[] args)  
{  
    Server app = new Server();  
    System.exit(app.main("Server", args));  
}  
}
```

The code imports the contents of the `Filesystem` package. This avoids having to continuously use fully-qualified identifiers with a `Filesystem.` prefix.

The next part of the source code is the definition of the `Server` class, which derives from `Ice.Application` and contains the main application logic in its `run` method. Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown in Figure 13.1.

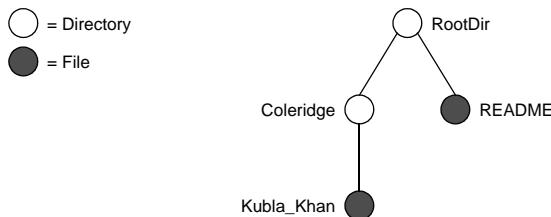


Figure 13.1. A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent.) Thus, the statement

```
DirectoryI root = new DirectoryI("/", null);
```

creates the root directory, with the name "/" and no parent directory.

Here is the code that establishes the structure in Figure 13.1:

```
// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI("/", null);

// Create a file "README" in the root directory
//
File file = new FileI("README", root);
String[] text;
text = new String[] {
    "This file system contains a collection of poetry."
};
try {
    file.write(text, null);
} catch (GenericError e) {
    System.err.println(e.reason);
}

// Create a directory "Coleridge" in the root directory
//
DirectoryI coleridge
```

```
        = new DirectoryI("Coleridge", root);

    // Create a file "Kubla_Khan" in the Coleridge directory
    //
    file = new FileI("Kubla_Khan", coleridge);
    text = new String[]{ "In Xanadu did Kubla Khan",
                         "A stately pleasure-dome decree:",
                         "Where Alph, the sacred river, ran",
                         "Through caverns measureless to man",
                         "Down to a sunless sea." };
    try {
        file.write(text, null);
    } catch (GenericError e) {
        System.err.println(e.reason);
    }
```

We first create the root directory and a file README within the root directory. (Note that we pass a reference to the root directory as the parent when we create the new node of type `FileI`.)

The next step is to fill the file with text:

```
String[] text;
text = new String[] {
    "This file system contains a collection of poetry."
};
try {
    file.write(text, null);
} catch (GenericError e) {
    System.err.println(e.reason);
}
```

Recall from Section 8.7.3 that Slice sequences by default map to Java arrays. The Slice type `Lines` is simply an array of strings; we add a line of text to our README file by initializing the `text` array to contain one element.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

```
file.write(text, null);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and *not* via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place—such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Java function call.

In similar fashion, the remainder of the code creates a subdirectory called Coleridge and, within that directory, a file called Kubla_Khan to complete the structure in Figure 13.1.

13.2.2 The FileI Servant Class

Our FileI servant class has the following basic structure:

```
public class FileI extends _FileDisp
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private String[] _lines;
}
```

The class has a number of data members:

- `_adapter`

This static member stores a reference to the single object adapter we use in our server.

- `_name`

This member stores the name of the file incarnated by the servant.

- `_parent`

This member stores the reference to the servant for the file's parent directory.

- `_lines`

This member holds the contents of the file.

The `_name` and `_parent` data members are initialized by the constructor:

```
public
FileI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    assert(_parent != null);

    // Create an identity
    //
    Ice.Identity myID
```

```
        = Ice.Util.stringToIdentity(Ice.Util.generateUUID());  
  
        // Add the identity to the object adapter  
        //  
        _adapter.add(this, myID);  
  
        // Create a proxy for the new node and  
        // add it as a child to the parent  
        //  
        NodePrx thisNode  
            = NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));  
        _parent.addChild(thisNode);  
    }  
}
```

After initializing the `_name` and `_parent` members, the code verifies that the reference to the parent is not null because every file must have a parent directory. The constructor then generates an identity for the file by calling `Ice.Util.generateUUID` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChild` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function on page 309.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

```
// Slice Node::name() operation  
  
public String  
name(Ice.Current current)  
{  
    return _name;  
}  
  
// Slice File::read() operation  
  
public String[]  
read(Ice.Current current)  
{  
    return _lines;  
}  
  
// Slice File::write() operation  
  
public void
```

```

        write(String[] text, Ice.Current current)
            throws GenericError
{
    _lines = text;
}

```

The `name` method is inherited from the generated `Node` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived). It simply returns the value of the `_name` member.

The `read` and `write` methods are inherited from the generated `File` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived) and simply return and set the `_lines` member.

13.2.3 The DirectoryI Servant Class

The `DirectoryI` class has the following basic structure:

```

package Filesystem;

public final class DirectoryI extends _DirectoryDisp
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private java.util.ArrayList _contents
        = new java.util.ArrayList();
}

```

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds a null reference.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

```

public
DirectoryI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    // Create an identity. The parent has the fixed identity "/"
    //
    Ice.Identity myID

```

```
        = Ice.Util.stringToIdentity(_parent != null ?
                                     Ice.Util.generateUUID() : "RootDir");

        // Add the identity to the object adapter
        //
        _adapter.add(this, myID);

        // Create a proxy for the new node and add it as a
        // child to the parent
        //
        NodePrx thisNode
            = NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
        if (_parent != null)
            _parent.addChild(thisNode);
    }
```

The constructor creates an identity for the new directory by calling `Ice.Util.generateUUID`. (For the root directory, we use the fixed identity "RootDir".) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a reference to itself and passes it to the `addChild` helper function.

`addChild` simply adds the passed reference to the `_contents` list:

```
void
addChild(NodePrx child)
{
    _contents.add(child);
}
```

The remainder of the operations, `name` and `list`, are trivial:

```
public String
name(Ice.Current current)
{
    return _name;
}

// Slice Directory::list() operation

public NodePrx[]
list(Ice.Current current)
{
    NodePrx[] result = new NodePrx[_contents.size()];
    _contents.toArray(result);
    return result;
}
```

Note that the `_contents` member is of type `java.util.ArrayList`, which is convenient for the implementation of the `addChild` method. However, this requires us to convert the list into a Java array in order to return it from the `list` operation.

13.3 Summary

This chapter showed how to implement a complete server for the file system we defined in Chapter 5. Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

Note that the server code we presented here is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the `_lines` data member while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, it is trivial to make the `read` and `write` operations thread-safe: a single data member and two lines of source code are sufficient to achieve this. We discuss how to write thread-safe servant implementations in Chapter 15.

Chapter 14

Ice Properties and Configuration

14.1 Chapter Overview

Ice uses a configuration mechanism that allows you to control many aspects of the behavior of your Ice applications at run time, such as maximum message size, number of threads, or whether to produce network trace messages. The configuration mechanism is not only useful to configure Ice, but you can also use it to provide configuration parameters to your own applications. The configuration mechanism is simple to use with a minimal API, yet flexible enough to cope with the needs of most applications.

Sections 14.2 to 14.7 describe the basics of the configuration mechanism and explain how to configure Ice via configuration files and command line options. Section 14.8 shows how you can create your own application-specific properties and how to access their values from within a program.

14.2 Properties

Ice and its various subsystems are configured by *properties*. A property is a name-value pair, for example:

```
Ice.UDP.SndSize=65535
```

In this example, the *property name* is `Ice.UDP.SndSize`, and the *property value* is `65535`.

You can find a complete list of the properties used to configure Ice in Appendix C.

14.2.1 Property Categories

By convention, Ice properties use the following naming scheme:

`<application>.<category>[.<sub-category>]`

Note that the sub-category is optional and not used by all Ice properties.

This two- or three-part naming scheme is by convention only—if you use properties to configure your own applications, you can use property names with any number of categories.

14.2.2 Reserved Prefixes

Ice reserves properties with the prefixes `Ice`, `IceBox`, `IcePack`, `IcePatch`, `IceSSL`, `IceStorm`, `Freeze`, and `Glacier`. You cannot use a property beginning with one of these prefixes to configure your own application.

14.2.3 Property Syntax

A property name consists of any number of non-white space characters, excluding the `#` and `=` characters. For example, the following are valid property names:

```
foo  
Foo  
foo.bar
```

Note that there is no special significance to a period in a property name. (Periods are used to make property names more readable and are not treated specially by the property parser.)

The following are invalid property names:

<code>foo bar</code>	<i>Illegal white space</i>
<code>foo=bar</code>	<i>Illegal =</i>
<code>foo#bar</code>	<i>Illegal #</i>

14.2.4 Value Syntax

A property value consists of any number of characters. Leading and trailing white space characters are ignored. Property values cannot contain a # character. The following are legal property values:

```
65535
yes
This is a = property value.
.../.../config
```

The following property value is illegal:

```
foo # bar      Property values cannot contain a # character
```

14.3 Configuration Files

Properties are usually set in a configuration file. A configuration file contains a number of name–value pairs, with each pair on a separate line. Empty lines and lines consisting entirely of white space characters are ignored. The # character introduces a comment that extends to the end of the current line.

Here is a simple configuration file:

```
# Example config file for Ice

Ice.MessageSizeMax = 2048      # Largest message size is 2MB
Ice.Trace.Network=3            # Highest level of tracing for network
Ice.Trace.Protocol=            # Disable protocol tracing
```

If you set the same property more than once, the last setting prevails and overrides any previous setting. Note that assigning nothing to a property clears that property.

For C++, Ice reads the contents of a configuration file when you create a communicator. By default, the name of the configuration file is determined by reading the contents of the **ICE_CONFIG** environment variable.¹ You can set this variable to a relative or absolute pathname of the configuration file, for example:

```
$ ICE_CONFIG=/opt/Ice/default_config
$ export ICE_CONFIG
$ ./server
```

1. The environment variable is not read by the Java run time; with Java, you must use the --Ice.Config property to set the name of the configuration file (see Section 14.5).

This causes the server to read its property settings from the configuration file in `/opt/Ice/default_config`.

14.4 Setting Properties on the Command Line

In addition to setting properties in a configuration file, you can also set properties on the command line, for example:

```
$ ./server --Ice.UDP.SndSize=65535 --IceSSL.Trace.Security=2
```

Any command line option that begins with `--` and is followed by one of the reserved prefixes (see page 312) is read and converted to a property setting when you create a communicator. Property settings on the command line override settings in a configuration file. If you set the same property more than once on the same command line, the last setting overrides any previous ones.

For convenience, any property not explicitly set to a value is set to the value 1. For example,

```
$ ./server --Ice.Trace.Protocol
```

is equivalent to

```
$ ./server --Ice.Trace.Protocol=1
```

Note that this feature only applies to properties that are set on the command line, but not to properties that are set from a configuration file.

You can also clear a property from the command line as follows:

```
$ ./server --Ice.Trace.Protocol=
```

As for properties set from a configuration file, assigning nothing to a property clears that property.

14.5 The `Ice.Config` Property

The `Ice.Config` property has special meaning to the Ice run time: it determines the pathname of a configuration file from which to read property settings. For example:

```
$ ./server --Ice.Config=/usr/local/filesystem/config
```

This causes property settings to be read from the configuration file in `/usr/local/filesystem/config`.

For C++, The `--Ice.Config` command-line option overrides any setting of the `ICE_CONFIG` environment variable, that is, if the `ICE_CONFIG` environment variable is set and you also use the `--Ice.Config` command-line option, the configuration file specified by the `ICE_CONFIG` environment variable is ignored.

If you use the `--Ice.Config` command-line option together with settings for other properties, the settings on the command line override the settings in the configuration file. For example:

```
$ ./server --Ice.Config=/usr/local/filesystem/config \
> --Ice.MessageSizeMax=4096
```

This sets the value of the `Ice.MessageSizeMax` property to 4096 regardless of any setting of this property in `/usr/local/filesystem/config`. The placement of the `--Ice.Config` option on the command line has no influence on this precedence. For example, the following command is equivalent to the preceding one:

```
$ ./server --Ice.MessageSizeMax=4096 \
> --Ice.Config=/usr/local/filesystem/config
```

Settings of the `Ice.Config` property inside a configuration file are ignored, that is, you can set `Ice.Config` only on the command line.

If you use the `--Ice.Config` option more than once, only the last setting of the option is used and the preceding ones are ignored. For example:

```
$ ./server --Ice.Config=file1 --Ice.Config=file2
```

This is equivalent to using:

```
$ ./server --Ice.Config=file2
```

You can use multiple configuration files by specifying a list of configuration file names, separated by commas. For example:

```
$ ./server --Ice.Config=/usr/local/filesystem/config,./config
```

This causes property settings to be retrieved from `/usr/local/filesystem/config`, followed by any settings in the file `config` in the current directory; settings in `./config` override settings `/usr/local/filesystem/config`. For C++, this mechanism also works for configuration files specified via the `ICE_CONFIG` environment variable.

14.6 Command-Line Parsing and Initialization

When you initialize the Ice run time by calling `Ice::initialize` (C++) or `Ice.Util.initialize` (Java), you pass an argument vector to the initialization call.

For C++, `Ice::initialize` accepts a C++ *reference* to `argc`:

```
namespace Ice {
    CommunicatorPtr initialize(int &argc, char *argv[]);
}
```

`Ice::initialize` parses the argument vector and initializes its property settings accordingly. In addition, it removes any arguments from `argv` that are property settings. For example, assume we invoke a server as:

```
$ ./server --myoption --Ice.Config=config -x a \
--Ice.Trace.Network=3 -y opt file
```

Initially, `argc` has the value 9, and `argv` has ten elements: the first nine elements contain the program name and the arguments, and the final element, `argv[argc]`, contains a null pointer (as required by the ISO C++ standard). When `Ice::initialize` returns, `argc` has the value 7 and `argv` contains the following elements:

```
. /server
--myoption
-x
a
-y
opt
file
0          # Terminating null pointer
```

This means that you should initialize the Ice run time before you parse the command line for your application-specific arguments. That way, the Ice-related options are stripped from the argument vector for you so you do not need to explicitly skip them. If you use the `Ice::Application` helper class (see Section 10.3.1), the `run` member function is passed the cleaned-up argument vector as well.

For Java, `Ice.Util.initialize` is overloaded. The signatures are:

```
package Ice;
public final class Util {

    public static Communicator
    initialize(String[] args);

    public static Communicator
    initialize(StringSeqHolder args);

    // ...
}
```

The first version does not strip Ice-related options for you, so, if you use that, you need to ignore options that start with one of the reserved prefixes (`--Ice`, `--IceBox`, `--IcePack`, `--IcePatch`, `--IceSSL`, `--IceStorm`, `--Freeze`, and `--Glacier`). The second version behaves like the C++ version and strips the Ice-related options from the passed argument vector.

If you use the `Ice.Application` helper class (see Section 12.4.1), the `run` method is passed the cleaned-up argument vector.

14.7 The `Ice.ProgramName` property

For C++, `initialize` sets the `Ice.ProgramName` property to the name of the current program (`argv[0]`). Ice uses the program name for log messages. Your application code can read this property and use it for similar purposes, for example, in diagnostic or trace messages. (See Section 14.8.1 for how to access the property value in your program.)

Even though `Ice.ProgramName` is initialized for you, you can still override its value from a configuration file or by setting the property on the command line.

For Java, the program name is not supplied as part of the argument vector—if you want the program name to appear in Ice log messages, you must set `Ice.ProgramName` before initializing a Java communicator.

14.8 Using Properties Programmatically

The Ice property mechanism is useful not only to configure Ice, but you can also use it as the configuration mechanism for your own applications. You can use the

same configuration file and command-line mechanism to set application-specific properties. For example, we could introduce a property to control the maximum file size for our file system application:

```
# Configuration file for file system application

Filesystem.MaxFileSize=1024      # Max file size in kB
```

The Ice run time stores the `Filesystem.MaxFileSize` property like any other property and makes it accessible via the `Properties` interface.

To access property values from within your program, you need to acquire the communicator's properties by calling `getProperties`:

```
module Ice {

    local interface Properties; // Forward declaration

    local interface Communicator {

        Properties getProperties();

        // ...
    };
}
```

The `Properties` interface provides methods to read and write property settings:

```
module Ice {
    local dictionary<string, string> PropertyDict;

    local interface Properties {

        string getProperty(string key);
        string getPropertyWithDefault(string key, string value);
        int getPropertyAsInt(string key);
        int getPropertyAsIntWithDefault(string key, int value);
        PropertyDict getPropertiesForPrefix(string prefix);

        void setProperty(string key, string value);

        StringSeq getCommandLineOptions();
        StringSeq parseCommandLineOptions(string prefix,
                                         StringSeq options);
        StringSeq parseIceCommandLineOptions(StringSeq options);

        void load(string file);
```

```
        Properties clone();
    };
};
```

14.8.1 Reading Properties

The operations to read property values behave as follows:

- `getProperty`

This operation returns the value of the specified property. If the property is not set, the operation returns the empty string.

- `getPropertyWithDefault`

This operation returns the value of the specified property. If the property is not set, the operation returns the supplied default value.

- `getPropertyAsInt`

This operation returns the value of the specified property as an integer. If the property is not set or contains a string that does not parse as an integer, the operation returns zero.

- `getPropertyAsIntWithDefault`

This operation returns the value of the specified property as an integer. If the property is not set or contains a string that does not parse as an integer, the operation returns the supplied default value.

- `getPropertiesForPrefix`

This operation returns all properties that begin with the specified prefix as a dictionary of type `PropertyDict`. This operation is useful if you want to extract the properties for a specific subsystem. For example,

```
getPropertiesForPrefix("Filesystem")
```

returns all properties that start with the prefix `Filesystem`, such as `Filesystem.MaxValue`. You can then use the usual dictionary lookup operations to extract the properties of interest from the returned dictionary.

With these lookup operations, using application-specific properties now becomes the simple matter of initializing a communicator as usual, getting access to the communicator's properties, and examining the desired property value. For example (in C++):

```
// ...  
  
Ice::CommunicatorPtr ic;  
  
// ...  
  
ic = Ice::initialize(argc, argv);  
  
// Get the maximum file size.  
//  
Ice::PropertiesPtr props = ic->getProperties();  
Ice::Int maxSize  
= props->getPropertyAsIntWithDefault("Filesystem.MaxFileSize",  
1024);  
  
// ...
```

Assuming that you have created a configuration file that sets the `Filesystem.MaxFileSize` property (and set the `ICE_CONFIG` variable or the `--Ice.Config` option accordingly), your application will pick up the configured value of the property.

14.8.2 Setting Properties

The `setProperty` operation sets a property to the specified value. (You can clear a property by setting it to the empty string.) This operation is useful only if you call it *before* you call `initialize`. This is because property values are (usually) read by the Ice run time only once, when you call `initialize`. The Ice run time does not guarantee that it will pay attention to a property value that is changed after you have initialized a communicator. Of course, this begs the question of how you can set a property value and have it also recognized by a communicator.

To permit you to set properties before initializing a communicator, the Ice run time provides an overloaded helper function called `getDefaultProperties`. In C++, the function is in the `Ice` namespace:

```
namespace Ice {  
    PropertiesPtr getDefaultProperties();  
    PropertiesPtr getDefaultProperties(int &argc, char *argv[]);  
};
```

`getDefaultProperties` returns a property set that is initialized with the contents of a configuration file as specified by either the `ICE_CONFIG` environ-

ment variable or, if you call the second version of the function, the **ICE_CONFIG** environment variable or the **--Ice.Config** option, overridden by any other property settings on the command line.

In Java, the function is a static method of the Util class inside the Ice package:

```
package Ice;

public final class Util {
    public static Properties getDefaultProperties();
    public static Properties getDefaultProperties(
        StringSeqHolder args);
    // ...
}
```

The first version returns an empty property set, and the second version returns a property set that is initialized with the **--Ice.Config** option, overridden by any other property settings on the command line.

`getDefaultProperties` is useful if you want to ensure that a property is set to a particular value, regardless of any setting of that property in a configuration file. For example:

```
// Get the initialized property set.
//
Ice::PropertiesPtr props = Ice::getDefaultProperties(argc, argv);

// Make sure that network and protocol tracing are off.
//
props->setProperty("Ice.Trace.Network", "0");
props->setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
Ice::CommunicatorPtr ic = Ice::initialize(argc, argv);

// ...
```

For Java, the equivalent code looks as follows:

```
Ice.StringSeqHolder argsH = new Ice.StringSeqHolder(args);
Ice.Properties properties = Ice.Util.getDefaultProperties(argsH);
properties.setProperty("Ice.Warn.Connections", "0");
communicator = Ice.Util.initialize(argsH);
```

Note that an extra step is required here: we first convert the argument array to an initialized `StringSeqHolder`. This is necessary so

`getDefaultProperties` can strip Ice-specific settings. In that way, we first obtain an initialized property set, then override the settings for the two tracing properties, and then pass the stripped argument vector to `initialize`.²

14.8.3 Parsing Properties

The Properties interface provides three operations to convert and parse properties:

- `getCommandLineOptions`

This operation converts an initialized set of properties into a sequence of equivalent command-line options. For example, if you have set the `Filesystem.MaxFileSize` property to 1024 and call `getCommandLineOptions`, the setting is returned as the string `"Filesystem.MaxFileSize=1024"`. This operation is useful for diagnostic purposes, for example, to dump the setting of all properties to a logging facility (see Section 16.14), or if you want to fork a new process with the same property settings as the current process.

- `parseCommandLineOptions`

This operation examines the passed argument vector for command-line options that have the specified prefix. Any options that match the prefix are converted to property settings (that is, they initialize the corresponding properties). The operation returns an argument vector that contains all those options that were *not* converted (that is, those options that did not match the prefix).

Because `parseCommandLineOptions` expects a sequence of strings, but C++ programs are used to dealing with `argc` and `argv`, Ice provides two utility functions that convert an `argc/argv` vector into a sequence of strings and vice-versa:

```
namespace Ice {  
  
    StringSeq argsToStringSeq(int argc, char* argv[]);
```

2. Recall from page 317 that, if we were to pass `args` directly to `getDefaultProperties` (without converting `args` to a `StringSeqHolder` first), `getDefaultProperties` would not strip Ice-related options, so the settings for the two tracing properties would not have any effect because the original argument array passed to `initialize` would override the explicit settings.

```
    void stringSeqToArgs(const StringSeq& args,
                         int& argc, char* argv[]);

}
```

You need to use `parseCommandLineOptions` (and the utility functions) if you want to permit application-specific properties to be set from the command line. For example, to permit the `--Filesystem.MaxFileSize` option to be used on the command line, we need to initialize our program as follows:

```
int
main(int argc, char * argv[])
{
    // Get the initialized property set.
    //
    Ice::PropertiesPtr props = Ice::getDefaultValueProperties();

    // Convert argc/argv to a string sequence.
    //
    Ice::StringSeq args = Ice::argsToStringSeq(argc, argv);

    // Strip out all options beginning with --Filesystem.
    //
    args = props->parseCommandLineOptions("Filesystem", args);

    // args now contains only those options that were not
    // stripped. Any options beginning with --Filesystem have
    // been converted to properties.

    // Convert remaining arguments back to argc/argv vector.
    //
    Ice::stringSeqToArgs(args, argc, argv);

    // Initialize communicator.
    //
    Ice::CommunicatorPtr ic = Ice::initialize(argc, argv);

    // At this point, argc/argv only contain options that
    // set neither an Ice property nor a Filesystem property,
    // so we can parse these options as usual.
    //
    // ...
}
```

Using this code, any options beginning with `--Filesystem` are converted to properties and are available via the property lookup operations as usual.

The call to `initialize` then removes any Ice-specific command-line options so, once the communicator is created, `argc/argv` only contains options and arguments that are not related to setting either a filesystem or an Ice property.

- `parseIceCommandLineOptions`

This operation behaves like `parseCommandLineOptions`, but removes the reserved Ice-specific options from the argument vector (see Section 14.2.2). It is used internally by the Ice run time to parse Ice-specific options in `initialize`.

14.8.4 Utility Operations

The `Properties` interface provides two utility operations:

- `clone`

This operation makes a copy of an existing property set. The copy contains exactly the same properties and values as the original. The operation is useful if you need to work with multiple property sets (see Section 14.8.5).

- `load`

This operation accepts a pathname to a configuration file and initializes the property set from that file. If the specified file cannot be read (for example, because it does not exist or the caller does not have read permission), the operation throws a `SyscallException`. This function is useful if you need to work with different configuration files for multiple property sets (see Section 14.8.5).

14.8.5 Working with Multiple Property Sets

Occasionally, you may need to work with multiple communicators and, hence, with multiple property sets. By default, `initialize` uses the property set that is returned by `getDefaultProperties`. This means that, if you create two communicators with `initialize`, they are created with identical property sets:

```
// Create a communicator.  
//  
Ice::CommunicatorPtr ic1 = initialize(argc, argv);  
  
// ...
```

```
// Create another communicator.  
Ice::CommunicatorPtr ic2 = initialize(argc, argv);  
  
// ic1 and ic2 are now initialized with the same property set.  
Internally, initialize uses a single, static property set. This means that, if you  
want to have two communicators with different property sets, the following  
attempt will not necessarily work:  
  
// Create a communicator.  
//  
Ice::CommunicatorPtr ic1 = initialize(argc, argv);  
  
// Set a property.  
//  
Ice::PropertiesPtr props = getDefaultProperties();  
props->setProperty("Ice.Trace.Network", "1");  
  
// Create another communicator.  
Ice::CommunicatorPtr ic2 = initialize(argc, argv);  
  
// ic1 and ic2 both may have tracing enabled!
```

The problem here is that the set returned by `getDefaultProperties` is internally shared by both communicators. For this example, both communicators end up with network tracing enabled. However, whether the setting of a particular property affects both communicators or just the second one depends on whether the property value is cached internally by the Ice run time: if the value is cached, only the second communicator is affected by the changed setting; if the value is not cached, both communicators are affected. In effect, the behavior of the preceding code is undefined.

To allow you to work with independent property sets, Ice provides utility functions to create a new property set. For C++, the functions are in the `Ice` namespace:

```
namespace Ice {  
    PropertiesPtr createProperties();  
    PropertiesPtr createProperties(int &argc, char *argv[]);  
}
```

The first version of `createProperties` creates a property set that is initialized with the contents of a configuration file as determined by reading the `ICE_CONFIG` environment variable. The second version creates a property set that is initialized with the contents of a configuration file as determined by the

ICE_CONFIG environment variable or the **--Ice.Config** command-line option.

For Java, the functions are provide in the `Util` class:

```
package Ice;
public final class Util {

    public static Properties createProperties();

    public static Properties createProperties(
        StringSeqHolder args);

    // ...
}
```

The first version of `createProperties` creates an empty property set. The second version creates a property set that is initialized the contents of a configuration file as determined by the **--Ice.Config** command-line option.

As an alternative to initializing a communicator, Ice provides the `initializeWithProperties` helper function. In C++, the declaration is:

```
namespace Ice {
    initializeWithProperties(int &argc, char *argv[],
                           const PropertiesPtr &props);
}
```

For Java, the `initializeWithProperties` is part of the `Ice.Util` class:

```
package Ice;
public final class Util {

    public static Communicator
    initializeWithProperties(String[] args, Properties properties)
    ;

    // ...
}
```

As you can see, `initializeWithProperties` allows you to explicitly pass a property set that is used in preference to the default property set. This allows you to create separate property sets for different communicators, for example:

```
// Create a property set for the first communicator.
// 
Ice::PropertiesPtr props1 = createProperties();

// Make sure that network tracing is off.
```

```
//  
props1->setProperty("Ice.Trace.Network", "0");  
  
// Initialize a communicator with this property set.  
//  
Ice::CommunicatorPtr ic1  
= Ice::initializeWithProperties(argc, argv, props1);  
  
// Create a property set for a second communicator.  
//  
Ice::PropertiesPtr props2 = createProperties();  
  
// Make sure that network tracing is on.  
//  
props2->setProperty("Ice.Trace.Network", "1");  
  
// Initialize a communicator with this property set.  
//  
Ice::CommunicatorPtr ic2  
= Ice::initializeWithProperties(argc, argv, props2);
```

The preceding code example correctly configures the two communicators with separate properties and so avoids the problems of the incorrect example on page 325. If you want to disallow command-line options to override property settings, you can pass a dummy argument vector to `initializeWithProperties`.

If you want to create communicators that differ in only a small number of properties, an alternative is to use the `clone` operation of a property set. The example that follows has the same effect as the preceding one. However, the first communicator is initialized using the default property set, whereas the second communicator is initialized with a copy of the default property set:

```
// Get the default property set for the first communicator.  
//  
Ice::PropertiesPtr props1 = getDefaultProperties(argc, argv);  
  
// Make sure that network tracing is off.  
//  
props1->setProperty("Ice.Trace.Network", "0");  
  
// Initialize a communicator with the default property set.  
//  
Ice::CommunicatorPtr ic1 = Ice::initialize(argv, argc);  
  
// Make a copy of the default property set.
```

```
//  
Ice::PropertiesPtr props2 = props1->clone();  
  
// Make sure that network tracing is on.  
//  
props2->setProperty("Ice.Trace.Network", "1");  
  
// Initialize a communicator with this property set.  
//  
Ice::CommunicatorPtr ic2  
= Ice::initializeWithProperties(argc, argv, props2);
```

If you want to use different configuration files for different communicators, you can use similar code but call the `load` operation (see Section 14.8.4) to initialize separate property sets that you have created with `createProperties`.

Note that you can retrieve the property set for a specific communicator at any time by calling `Communicator::getProperties`.

14.9 Summary

The Ice property mechanism provides a simple way to configure Ice by setting properties in configuration files or on the command line. This also applies to your own applications: you can easily use the `Properties` interface to access application-specific properties that you have created for your own needs. The API to access property values is small and simple, making it easy to retrieve property values at run time, yet is flexible enough to allow you to work with different property sets and configuration files if the need arises.

Chapter 15

Threads and Concurrency with C++

15.1 Chapter Overview

This chapter presents the C++ threading and signal handling abstractions that are provided by Ice. (Note that there is no equivalent threading API for Java—for Java, use the built-in threading and synchronization facilities.) We briefly outline the threading model used by Ice and how to use each of the available synchronization primitives (mutexes and monitors). We then cover how to create, control, and destroy threads. The threading discussion concludes with a brief example that shows how to create a thread-safe producer-consumer application that uses several threads. Finally, we introduce a portable abstraction for handling signals and signal-like events.

15.2 Introduction

Ice is inherently a multi-threaded platform. There is no such thing as a single-threaded server in Ice. As a result, you must concern yourself with concurrency issues: if a thread reads a data structure while another thread updates the same data structure, havoc will ensue unless you protect the data structure with appropriate locks.

Threading and concurrency control vary widely with different operating systems. To make threads programming easier and portable, Ice provides a simple thread abstraction layer that allows you to write portable source code regardless of the underlying platform. In this chapter, we take a closer look at the threading and concurrency control mechanisms in Ice.

Note that we assume that you are familiar with light-weight threads and concurrency control. (See [8] for an excellent treatment of programming with threads.)

15.3 The Ice Threading Model

An Ice server is inherently multi-threaded. The server-side run time maintains a thread pool that processes incoming requests. Each operation invocation received from clients is dispatched in its own thread, using a leader-follower threading model (see [17]). If the thread pool is exhausted because all threads are currently executing an operation invocation in the server, additional incoming client requests are transparently delayed until an operation completes in the server and relinquishes its thread; that thread is then used to dispatch the next pending client request.

The size of the thread pool in the server is configurable via the `Ice.ThreadPool.Server.Size` property (see Appendix C). The default value for this property is 1.

Threading means that multiple invocations from clients can be executing concurrently in the server. In fact, it is possible for multiple requests to proceed in parallel within the same servant and within the same operation on that servant. It follows that, if the operation implementation manipulates non-stack storage (such as member variables of the servant or global or static data), you must interlock access to this data to avoid data corruption. The Ice threads library provides a number of synchronization primitives, such as simple mutexes, read-write locks, and monitors. These synchronization primitives permit you to implement concurrency control at different levels of granularity. In addition, Ice also permits you create threads of your own. This allows you to, for example, create a separate thread that can respond to GUI or other asynchronous events.

15.4 Library Overview

The Ice threading library provides the following thread-related abstractions:

- mutexes
- recursive mutexes
- read-write recursive mutexes
- monitors
- a thread abstraction that allows you to create, control, and destroy threads

All the threading APIs are part of the `IceUtil` namespace.

15.5 Mutexes

The classes `IceUtil::Mutex` (defined in `IceUtil/Mutex.h`) and `IceUtil::StaticMutex` (defined in `IceUtil/StaticMutex.h`) provide simple non-recursive mutual exclusion mechanisms:

```
namespace IceUtil {

    class Mutex {
    public:
        Mutex();
        ~Mutex();
        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<Mutex> Lock;
        typedef TryLockT<Mutex> TryLock;
    };

    struct StaticMutex {
        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<StaticMutex> Lock;
        typedef TryLockT<StaticMutex> TryLock;
    };
}
```

`IceUtil::Mutex` and `IceUtil::StaticMutex` have identical behavior, but `IceUtil::StaticMutex` is implemented as a simple data structure¹ so that instances can be declared statically and initialized during compilation, as demonstrated below.

```
static IceUtil::StaticMutex myStaticMutex =
    ICE_STATIC_MUTEX_INITIALIZER;
```

The preprocessor macro `ICE_STATIC_MUTEX_INITIALIZER` is defined to correctly initialize the data members of `IceUtil::StaticMutex`. Instances of `IceUtil::StaticMutex` are never destroyed.

`IceUtil::Mutex`, on the other hand, is implemented as a class and therefore is initialized by its constructor and destroyed by its destructor.

The member functions of these classes work as follows:

- `lock`

The `lock` function attempts to acquire the mutex. If the mutex is already locked, it suspends the calling thread until the mutex becomes available. The call returns once the calling thread has acquired the mutex.

- `tryLock`

The `tryLock` function attempts to acquire the mutex. If the mutex is available, the call returns with the mutex locked and returns `true`. Otherwise, if the mutex is locked by another thread, the call returns `false`.

- `unlock`

The `unlock` function unlocks the mutex.

Note that `IceUtil::Mutex` and `IceUtil::StaticMutex` are non-recursive mutex implementations. This means that you must adhere to the following rules:

- Do not call `lock` on the same mutex more than once from a thread. The mutex is not recursive so, if the owner of a mutex attempts to lock it a second time, the behavior is undefined.
- Do not call `unlock` on a mutex unless the calling thread holds the lock. Calling `unlock` on a mutex that is not currently held by any thread, or calling `unlock` on a mutex that is held by a different thread, results in undefined behavior.

1. In ISO C++ terminology, `StaticMutex` is “plain old data” (POD).

15.5.1 Thread-Safe File Access for the Filesystem Application

Recall that the implementation of the `read` and `write` operations for our file system server in Section 11.2.3 is not thread safe:

```
Filesystem::Lines  
Filesystem::FileI::read(const Ice::Current &) const  
{  
    return _lines;           // Not thread safe!  
}  
  
void  
Filesystem::FileI::write(const Filesystem::Lines & text,  
                        const Ice::Current &)  
{  
    _lines = text;          // Not thread safe!  
}
```

The problem here is that, if we receive concurrent invocations of `read` and `write`, one thread will be assigning to the `_lines` vector while another thread is reading that same vector. The outcome of such concurrent data access is undefined; to avoid the problem, we need to serialize access to the `_lines` member with a mutex. We can make the mutex a data member of the `FileI` class and lock and unlock it in the `read` and `write` operations:

```
#include <IceUtil/Mutex.h>  
// ...  
  
namespace Filesystem {  
    // ...  
  
    class FileI : virtual public File,  
                  virtual public Filesystem::NodeI {  
public:  
    // As before...  
private:  
    Lines _lines;  
    IceUtil::Mutex _fileMutex;  
};  
// ...  
}  
  
Filesystem::Lines  
Filesystem::FileI::read(const Ice::Current &) const  
{  
    _fileMutex.lock();
```

```

        Lines l = _lines;
        _fileMutex.unlock();
        return l;
    }

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    _fileMutex.lock();
    _lines = text;
    _fileMutex.unlock();
}

```

The `FileI` class here is identical to the implementation in Section 11.2.2, except that we have added the `_fileMutex` data member. The `read` and `write` operations lock and unlock the mutex to ensure that only one thread can read or write the file at a time. Note that, by using a separate mutex for each `FileI` instance, it is still possible for multiple threads to concurrently read or write files, as long as they each access a *different* file. Only concurrent accesses to the *same* file are serialized.

The implementation of `read` is somewhat awkward here: we must make a local copy of the file contents while we are holding the lock and return that copy. Doing so is necessary because we must unlock the mutex before we can return from the function. However, as we will see in the next section, the copy can be avoided by using a helper class that unlocks the mutex automatically when the function returns.

15.5.2 Guaranteed Unlocking of Mutexes

Using the raw `lock` and `unlock` operations on mutexes has an inherent problem: if you forget to unlock a mutex, your program will deadlock. Forgetting to unlock a mutex is easier than you might suspect, for example:

```

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    _fileMutex.lock();           // Lock the mutex
    Lines l = readFileContents(); // Read from database
    _fileMutex.unlock();         // Unlock the mutex
    return l;
}

```

Assume that we are keeping the contents of the file on secondary storage, such as a database, and that the `readFileContents` function accesses the file. The code is almost identical to the previous example but now contains a latent bug: if `readFileContents` throws an exception, the `read` function terminates without ever unlocking the mutex. In other words, this implementation of `read` is not exception-safe.

The same problem can easily arise if you have a larger function with multiple return paths. For example:

```
void
SomeClass::someFunction(/* params here... */)
{
    _mutex.lock();                                // Lock a mutex

    // Lots of complex code here...

    if (someCondition) {
        // More complex code here...
        return;                                     // Oops!!!
    }

    // More code here...

    _mutex.unlock();                               // Unlock the mutex
}
```

In this example, the early return from the middle of the function leaves the mutex locked. Even though this example makes the problem quite obvious, in large and complex pieces of code, both exceptions and early returns can cause hard-to-track deadlock problems. To avoid this, the `Mutex` class contains two type definitions for helper classes, called `Lock` and `TryLock`:

```
namespace IceUtil {

    class Mutex {
        // ...

        typedef LockT<Mutex> Lock;
        typedef TryLockT<Mutex> TryLock;
    };
}
```

`LockT` and `TryLockT` are simple templates that primarily consist of a constructor and a destructor; the constructor calls `lock` on its argument, and the

destructor calls `unlock`. By instantiating a local variable of type `Lock` or `TryLock`, we can avoid the deadlock problem entirely:

```
void
SomeClass::someFunction(/* params here... */)
{
    IceUtil::Mutex::Lock lock(_mutex); // Lock a mutex

    // Lots of complex code here...

    if (someCondition) {
        // More complex code here...
        return; // No problem
    }

    // More code here...
}

} // Destructor of lock unlocks the mutex
```

On entry to `someFunction`, we instantiate a local variable `lock`, of type `IceUtil::Mutex::Lock`. The constructor of `lock` calls `lock` on the mutex so the remainder of the function is inside a critical region. Eventually, `someFunction` returns, either via an ordinary return (in the middle of the function or at the end) or because an exception was thrown somewhere in the function body. Regardless of how the function terminates, the C++ run time unwinds the stack and calls the destructor of `lock`, which unlocks the mutex, so we cannot get trapped by the deadlock problem we had previously.

You should make it a habit to always use the `Lock` and `TryLock` helpers instead of calling `lock` and `unlock` directly. Doing so results in code that is easier to understand and maintain.

Using the `Lock` helper, we can rewrite the implementation of our `read` and `write` operations as follows:

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                       const Ice::Current &)
```

```
{  
    IceUtil::Mutex::Lock lock(_fileMutex);  
    _lines = text;  
}
```

Note that this also eliminates the need to make a copy of the `_lines` data member: the return value is initialized under protection of the mutex and cannot be modified by another thread once the destructor of `lock` unlocks the mutex.

15.6 Recursive Mutexes

As we saw on page 332, a non-recursive mutex cannot be locked more than once, even by the thread that holds the lock. This frequently becomes a problem if a program contains a number of functions, each of which must acquire a mutex, and you want to call one function as part of the implementation of another function:

```
IceUtil::Mutex _mutex;  
  
void  
f1()  
{  
    IceUtil::Mutex::Lock lock(_mutex);  
    // ...  
}  
  
void  
f2()  
{  
    IceUtil::Mutex::Lock lock(_mutex);  
    // Some code here...  
  
    // Call f1 as a helper function  
    f1();  
    // Deadlock!  
  
    // More code here...
}
```

`f1` and `f2` each correctly lock the mutex before manipulating data but, as part of its implementation, `f2` calls `f1`. At that point, the program deadlocks because `f2` already holds the lock that `f1` is trying to acquire. For this simple example, the problem is obvious. However, in complex systems with many functions that acquire and release locks, it can get very difficult to track down this kind of situation: the locking conventions are not manifest anywhere but in the source code

and each caller must know which locks to acquire (or not to acquire) before calling a function. The resulting complexity can quickly get out of hand.

Ice provides a recursive mutex class `RecMutex` (defined in `IceUtil/RecMutex.h`) that avoids this problem:

```
namespace IceUtil {  
  
    class RecMutex {  
    public:  
        void lock() const;  
        bool tryLock() const;  
        void unlock() const;  
  
        typedef LockT<RecMutex> Lock;  
        typedef TryLockT<RecMutex> TryLock;  
    };  
}
```

Note that the signatures of the operations are the same as for `IceUtil::Mutex`. However, `RecMutex` implements a recursive mutex:

- `lock`

The `lock` function attempts to acquire the mutex. If the mutex is already locked by another thread, it suspends the calling thread until the mutex becomes available. If the mutex is available or is already locked by the calling thread, the call returns immediately with the mutex locked.

- `tryLock`

The `tryLock` function works like `lock`, but, instead of blocking the caller, it returns `false` if the mutex is locked by another thread. Otherwise, the return value is `true`.

- `unlock`

The `unlock` function unlocks the mutex.

As for non-recursive mutexes, you must adhere to a few simple rules for recursive mutexes:

- Do not call `unlock` on a mutex unless the calling thread holds the lock.
- You must call `unlock` as many times as you called `lock` for the mutex to become available to another thread. (Internally, a recursive mutex is implemented with a counter that is initialized to zero. Each call to `lock` increments the counter and each call to `unlock` decrements the counter; the mutex is made available to another thread when the counter returns to zero.)

Using recursive mutexes, the code fragment on page 337 works correctly:

```
#include <IceUtil/RecMutex.h>
// ...

IceUtil::RecMutex _mutex;           // Recursive mutex

void
f1()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // ...
}

void
f2()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // Some code here...

    // Call f1 as a helper function
    f1();                         // Fine

    // More code here...
}
```

Note that the type of the mutex is now `RecMutex` instead of `Mutex`, and that we are using the `Lock` type definition provided by the `RecMutex` class, not the one provided by the `Mutex` class.

15.7 Read-Write Recursive Mutexes

The implementation of our `read` and `write` operations on page 336 is more conservative in its locking than strictly necessary: only one thread can be in either the `read` or `write` operation at a time. However, we have problems with concurrent file access only if we have concurrent writers, or concurrent readers and writers for the same file. However, if we have only readers, there is no need to serialize access for all the reading threads because none of them updates the file contents.

Ice provides a read-write recursive mutex class `RWRecMutex` (defined in `IceUtil/RWRecMutex.h`) that implements a reader-writer lock:

```
namespace IceUtil {

    class RWRecMutex {
public:
    void readLock() const;
    bool tryReadLock() const;
    bool timedReadLock(const Time &) const;

    void writeLock() const;
    bool tryWriteLock() const;
    bool timedWriteLock(const Time &) const;

    void unlock() const;

    void upgrade() const;
    bool timedUpgrade(const Time &) const;

    typedef RLockT<RWRecMutex> RLock;
    typedef TryRLockT<RWRecMutex> TryRLock;
    typedef WLockT<RWRecMutex> WLock;
    typedef TryWLockT<RWRecMutex> TryWLock;
};

}
```

A read-write recursive mutex splits the usual single lock operation into `readLock` and `writeLock` operations. Multiple readers can each acquire the mutex in parallel. However, only a single writer can hold the mutex at any one time (with neither other readers nor other writers being present). A `RWRecMutex` is recursive, meaning that you can call `readLock` or `writeLock` multiple times from the same calling thread.

The member functions behave as follows:

- `readLock`

This function acquires a read lock. If a writer currently holds the mutex, the caller is suspended until the mutex becomes available for reading. If the mutex is available, or only readers currently hold the mutex, the call returns immediately with the mutex locked.

- `tryReadLock`

This function attempts to acquire a read lock. If the lock is currently held by a writer, the function returns `false`. Otherwise, it acquires the lock and returns `true`.

- `timedReadLock`

This function attempts to acquire a read lock. If the lock is currently held by a writer, the function waits for the specified timeout. If the lock can be acquired within the timeout, the function returns `true` with the lock held. Otherwise, once the timeout expires, the function returns `false`. (See Section 15.8 for how to construct a timeout value.)

- `writeLock`

This function acquires a write lock. If readers or a writer currently hold the mutex, the caller is suspended until the mutex becomes available for writing. If the mutex is available, the call returns immediately with the lock held.

- `tryWriteLock`

This function attempts to acquire a write lock. If the lock is currently held by readers or a writer, the function returns `false`. Otherwise, it acquires the lock and returns `true`.

- `timedWriteLock`

This function attempts to acquire a write lock. If the lock is currently held by readers or a writer, the function waits for the specified timeout. If the lock can be acquired within the timeout, the function returns `true` with the lock held. Otherwise, once the timeout expires, the function returns `false`. (See Section 15.8 for how to construct a timeout value.)

- `unlock`

This function unlocks the mutex (whether currently held for reading or writing).

- `upgrade`

This function upgrades a read lock to a write lock. If other readers currently hold the mutex, the caller is suspended until the mutex becomes available for writing. If the mutex is available, the call returns immediately with the lock held.

Note that `upgrade` is non-recursive. Do not call it more than once from the same thread.

- `timedUpgrade`

This function attempts to upgrade a read lock to a write lock. If the lock is currently held by other readers, the function waits for the specified timeout. If the lock can be acquired within the timeout, the function returns `true` with

the lock held. Otherwise, once the timeout expires, the function returns `false`. (See Section 15.8 for how to construct a timeout value.)

Note that `timedUpgrade` is non-recursive. Do not call it more than once from the same thread.

As for non-recursive and recursive mutexes, you must adhere to a few rules for correct use of read-write locks:

- Do not call `unlock` on a mutex unless the calling thread holds the lock.
- You must call `unlock` as many times as you called `readLock` or `writeLock` for the mutex to become available to another thread.
- Do not call `upgrade` or `timedUpgrade` on a mutex for which you do not hold a read lock.
- `upgrade` and `timedUpgrade` are non-recursive (because making them recursive would incur an unacceptable performance penalty). Do not call these methods more than once from the same thread.

The implementation of read-write recursive mutexes gives preference to writers: if a writer is waiting to acquire the lock, no new readers are permitted to acquire the lock; the implementations waits until all current readers relinquish the lock and then locks the mutex for the waiting writer. Note that mutexes do not implement any notion of fairness: if multiple writers are continuously waiting to acquire a write lock, which writer gets the lock next depends on the underlying threads implementation. There is no queue of waiting writers to ensure that none of the writers are permanently starved of access to the mutex.

Using a `RWRecMutex`, we can implement our read and write operations to allow multiple readers in parallel, or a single writer:

```
#include <IceUtil/RWRecMutex.h>
// ...

namespace Filesystem {
    // ...

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
public:
    // As before...
private:
    Lines _lines;
    IceUtil::RWRecMutex _fileMutex; // Read-write mutex
};
```

```
}

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    IceUtil::RWRecMutex::RLock lock(_fileMutex); // Read lock
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    IceUtil::RWRecMutex::WLock lock(_fileMutex); // Write lock
    _lines = text;
}
```

This code is almost identical to the non-recursive version on page 333. Note that the only changes are that we have changed the type of the mutex in the servant to `RWRecMutex` and that we are using the `RLock` and `WLock` helpers to guarantee unlocking instead of calling `readLock` and `writeLock` directly.

15.8 Timed Locks

As we saw on page 340, read-write locks provide member functions that operate with a timeout. The amount of time to wait is specified by an instance of the `IceUtil::Time` class (defined in `IceUtil/Time.h`):

```
namespace IceUtil {

    class Time {
public:
    Time();
    static Time now();
    static Time seconds(long);
    static Time milliseconds(long);
    static Time microseconds(long long);

    Time operator-() const;
    Time operator-(const Time&) const;
    Time operator+(const Time&) const;
    Time& operator-=(const Time&);
    Time& operator+=(const Time&);
```

```
    bool operator<(const Time&) const;
    bool operator<=(const Time&) const;
    bool operator>(const Time&) const;
    bool operator>=(const Time&) const;
    bool operator==(const Time&) const;
    bool operator!=(const Time&) const;

    operator timeval() const;
    operator double() const;
};

}
```

The `Time` class provides basic facilities for getting the current time, constructing time intervals, adding and subtracting times, and comparing times:

- `Time`

Internally, the `Time` class stores ticks in microsecond units. For absolute time, this is the number of microseconds since the UNIX epoch (00:00:00 UTC on 1 Jan. 1970). For durations, this is the number of microseconds in the duration. The default constructor initializes the tick count to zero.

- `now`

This function constructs a `Time` object that is initialized to the current time of day.

- `seconds`

`milliSeconds`
`microSeconds`

These functions construct `Time` objects from the argument in the specified units. For example, the following code fragment creates a time duration of one minute:

```
IceUtil::Time t = IceUtil::Time::seconds(60);
```

- `operator-`

`operator+`

`operator-=`

`operator+=`

These operators allow you to add and subtract times from each other. For example:

```
IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
IceUtil::Time oneMinuteAgo = IceUtil::Time::now() - oneMinute;
```

- The comparison operators allow you compare times and time intervals with each other, for example:

```
IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
IceUtil::Time twoMinutes = IceUtil::Time::seconds(120);
assert(oneMinute < twoMinutes);
```

- **operator timeval**

This operator converts a Time object to a `struct timeval`, defined as follows:

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

The conversion is useful for API calls that require a `struct timeval` argument, such as `select`. To convert a duration into a `timeval` structure, simply assign a Time object to a `struct timeval`:

```
IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
struct timeval tv;
tv = t;
```

- **operator double**

This operator converts a Time object into a double value that expresses time in seconds:

```
IceUtil::Time t = IceUtil::Time::milliseconds(500);
double d = t;
cerr << "Duration: " << d << " seconds" << endl;
```

Running this code results in the output:

```
Duration: 0.5 seconds
```

Using a Time object with a timed lock operation is trivial, for example:

```
#include <IceUtil/RWRecMutex.h>

// ...
IceUtil::RWRecMutex _mutex;

// ...

try {
    // Wait for up to two seconds to get a write lock...
    //
```

```

IceUtil::RWRecMutex::TryWLock
    lock(_mutex, IceUtil::Time::seconds(2));

    // Got the lock -- destructor of lock will unlock

} catch (const IceUtil::ThreadLockedException &) {

    // Waited for two seconds without getting the lock...
}

```

Note that the TryRLock and TryWLock constructors are overloaded: if you supply only a mutex as the sole argument, the constructor calls `tryReadLock` or `tryWriteLock`; if you supply both a mutex and a timeout, the constructor calls `timedReadLock` or `timedWriteLock`.

15.9 Monitors

Mutexes implement a simple mutual exclusion mechanism that allows only a single thread (or, in the case of read-write mutexes, a single writer thread or multiple reader threads) to be active in a critical region at a time. In particular, for another thread to enter the critical region, another thread must leave it. This means that, with mutexes, it is impossible to suspend a thread inside a critical region and have that thread wake up again at a later time, for example, when a condition becomes true.

To address this problem, Ice provides a monitor. Briefly, a monitor is a synchronization mechanism that protects a critical region: as for a mutex, only one thread may be active at a time inside the critical region. However, a monitor allows you to suspend a thread inside the critical region; doing so allows another thread to enter the critical region. The second thread can either leave the monitor (thereby unlocking the monitor), or it can suspend itself inside the monitor; either way, the original thread is woken up and continues execution inside the monitor. This extends to any number of threads, so several threads can be suspended inside a monitor.²

2. The monitors provided by Ice have *Mesa* semantics, so called because they were first implemented by the Mesa programming language [12]. Mesa monitors are provided by a number of languages, including Java and Ada. With Mesa semantics, the signalling thread continues to run and another thread gets to run only once the signalling thread suspends itself or leaves the monitor.

Monitors provide a more flexible mutual exclusion mechanism than mutexes because they allow a thread to check a condition and, if the condition is false, put itself to sleep; the thread is woken up by some other thread that has changed the condition.

15.9.1 The Monitor Class

Ice provides monitors with the `IceUtil::Monitor` class (defined in `IceUtil/Monitor.h`):

```
namespace IceUtil {

    template <class T>
    class Monitor {
    public:
        void lock() const;
        void unlock() const;
        bool tryLock() const;

        void wait() const;
        bool timedWait(const Time&) const;
        void notify();
        void notifyAll();

        typedef LockT<Monitor<T> > Lock;
        typedef TryLockT<Monitor<T> > TryLock;
    };
}
```

Note that `Monitor` is a template class that requires either `Mutex` or `RecMutex` as its template parameter. (Instantiating a `Monitor` with a `RecMutex` makes the monitor recursive.)

The member functions behave as follows:

- `lock`

This function attempts to lock the monitor. If the monitor is currently locked by another thread, the calling thread is suspended until the monitor becomes available. The call returns with the monitor locked.

- `tryLock`

This function attempts to lock a monitor. If the monitor is available, the call returns `true` with the monitor locked. If the monitor is locked by another thread, the call returns `false`.

- **unlock**

This function unlocks a monitor. If other threads are waiting to enter the monitor (are blocked inside a call to `lock`), one of the threads is woken up and locks the monitor.

- **wait**

This function suspends the calling thread and, at the same time, release the lock on the monitor. A thread suspended inside a call to `wait` can be woken up by another thread that calls `notify` or `notifyAll`. When the call returns, the suspended thread resumes execution with the monitor locked.

- **timedWait**

This function suspends the calling thread for up to the specified timeout. If another thread calls `notify` or `notifyAll` and wakes up the suspended thread before the timeout expires, the call returns `true` and the suspended thread resumes execution with the monitor locked. Otherwise, if the timeout expires, the function returns `false`.

- **notify**

This function wakes up a single thread that is currently suspended in a call to `wait`. If no thread is suspended in a call to `wait` at the time `notify` is called, the notification is lost (that is, calls to `notify` are *not* remembered if there is no thread to be woken up).

Note that notifying does not run another thread immediately. Another thread gets to run only once the notifying thread either calls `wait` or unlocks the monitor (Mesa semantics).

- **notifyAll**

This function wakes up all threads that are currently suspended in a call to `wait`. As for `notify`, calls to `notifyAll` are lost if no threads are suspended at the time.

As for `notify`, `notifyAll` causes other threads to run only once the notifying thread has either called `wait` or unlocked the monitor (Mesa semantics).

You must adhere to a few rules for monitors to work correctly:

- Do not call `unlock` unless you hold the lock. If you instantiate a monitor with a recursive mutex, you get recursive semantics, that is, you must call `unlock` as many times as you have called `lock` (or `tryLock`) for the monitor to become available.

- Do not call `wait` or `timedWait` unless you hold the lock.
- Do not call `notify` or `notifyAll` unless you hold the lock.
- When returning from a `wait` call, you *must* re-test the condition before proceeding (see page 351).

15.9.2 Using Monitors

To illustrate how to use a monitor, consider a simple unbounded queue of items. A number of producer threads add items to the queue, and a number of consumer threads remove items from the queue. If the queue becomes empty, consumers must wait until a producer puts a new item on the queue. The queue itself is a critical region, that is, we cannot allow a producer to put an item on the queue while a consumer is removing an item. Here is a very simple implementation of a such a queue:

```
template<class T> class Queue {
public:
    void put(const T & item) {
        _q.push_back(item);
    }

    T get() {
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

As you can see, producers call the `put` method to enqueue an item, and consumers call the `get` method to dequeue an item. Obviously, this implementation of the queue is not thread-safe and there is nothing to stop a consumer from attempting to dequeue an item from an empty queue.

Here is a version of the queue that uses a monitor to suspend a consumer if the queue is empty:

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
```

```

void put(const T & item) {
    IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
    _q.push_back(item);
    notify();
}

T get() {
    IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
    while (_q.size() == 0)
        wait();
    T item = _q.front();
    _q.pop_front();
    return item;
}

private:
    list<T> _q;
};

```

Note that the `Queue` class now inherits from

`IceUtil::Monitor<IceUtil::Mutex>`, that is, `Queue` *is-a* monitor.

Both the `put` and `get` methods lock the monitor when they are called. As for mutexes, instead of calling `lock` and `unlock` directly, we are using the `Lock` helper which automatically locks the monitor when it is instantiated and unlocks the monitor again when it is destroyed.

The `put` method first locks the monitor and then, now being in sole possession of the critical region, enqueues an item. Before returning (thereby unlocking the monitor), `put` calls `notify`. The call to `notify` will wake up any consumer thread that may be asleep in a `wait` call to inform the consumer that an item is available.

The `get` method also locks the monitor and then, before attempting to dequeue an item, tests whether the queue is empty. If so, the consumer calls `wait`. This suspends the consumer inside the `wait` call and unlocks the monitor, so a producer can enter the monitor to enqueue an item. Once that happens, the producer calls `notify`, which causes the consumer's `wait` call to complete, with the monitor again locked for the consumer. The consumer now dequeues an item and returns (thereby unlocking the monitor).

For this machinery to work correctly, the implementation of `get` does two things:

- `get` tests whether the queue is empty *after* acquiring the lock.

- `get` re-tests the condition in a loop around the call to `wait`; if the queue is still empty after `wait` returns, the `wait` call is re-entered.

You *must* always write your code to follow the same pattern:

- *Never* test a condition unless you hold the lock.
- *Always* re-test the condition in a loop around `wait`. If the test still shows the wrong outcome, call `wait` again.

Not adhering to these conditions will eventually result in a thread accessing shared data when it is not in its expected state, for the following reasons:

1. If you test a condition without holding the lock, there is nothing to prevent another thread from entering the monitor and changing its state before you can acquire the lock. This means that, by the time you get around to locking the monitor, the state of the monitor may no longer be in agreement with the result of the test.
2. Some thread implementations suffer from a problem known as *spurious wake-up*: occasionally, more than one thread may wake up in response to a call to `notify`. As a result, each thread that returns from a call to `wait` must re-test the condition to ensure that the monitor is in its expected state: the fact that `wait` returns does *not* indicate that the condition has changed.

15.9.3 Efficient Notification

The previous implementation of our thread-safe queue on page 349 unconditionally notifies a waiting reader whenever a writer deposits an item into the queue. If no reader is waiting, the notification is lost and does no harm. However, unless there is only a single reader and writer, many notifications will be sent unnecessarily, caused unwanted overhead.

Here is one way to fix the problem:

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T & item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_q.size() == 1)
            notify();
    }
}
```

```

T get() {
    IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
    while (_q.size() == 0)
        wait();
    T item = _q.front();
    _q.pop_front();
    return item;
}

private:
    list<T> _q;
};

```

The only difference between this code and the implementation on page 349 is that a writer calls `notify` only if the queue length has just changed from empty to non-empty. That way, unnecessary `notify` calls are never made. However, this approach works only for a single reader thread. To see why, consider the following scenario:

1. Assume that the queue currently contains a number of items and that we have five reader threads.
2. The five reader threads continue to call `get` until the queue becomes empty and all five readers are waiting in `get`.
3. The scheduler schedules a writer thread. The writer finds the queue empty, deposits an item, and wakes up a single reader thread.
4. The awakened reader thread dequeues the single item on the queue.
5. The reader calls `get` a second time, finds the queue empty, and goes to sleep again.

The net effect of this is that there is a good chance that only one reader thread will ever be active; the other four reader threads end up being permanently asleep inside the `get` method.

One way around this problem is call `notifyAll` instead of `notify` once the queue length exceeds a certain amount, for example:

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T & item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
    }
}
```

```
        if (_q.size() >= _wakeupThreshold)
            notifyAll();
    }

T get() {
    IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
    while (_q.size() == 0)
        wait();
    T item = _q.front();
    _q.pop_front();
    return item;
}

private:
    list<T> _q;
    const int _wakeupThreshold = 100;
};
```

Here, we have added a private data member `_wakeupThreshold`; a writer wakes up *all* waiting readers once the queue length exceeds the threshold, in the expectation that all the readers will consume items more quickly than they are produced, thereby reducing the queue length below the threshold again.

This approach works, but has drawbacks as well:

- The appropriate value of `_wakeupThreshold` is difficult to determine and sensitive to things such as speed and number of processors and I/O bandwidth.
- If multiple readers are asleep, they are all made runnable by the thread scheduler once a writer calls `notifyAll`. On a multiprocessor machine, this may result in all readers running at once (one per CPU). However, as soon as the readers are made runnable, each of them attempts to reacquire the mutex that protects the monitor before returning from `wait`. Of course, only one of the readers actually succeeds and the remaining readers are suspended again, waiting for the mutex to become available. The net result is a large number of thread context switches as well as repeated and unnecessary locking of the system bus.

A better option than calling `notifyAll` is to wake up waiting readers one at a time. To do this, we keep track of the number of waiting readers and call `notify` only if a reader needs to be woken up:

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
```

```

public:
    Queue() : _waitingReaders(0) {}

    void put(const T & item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_waitingReaders)
            notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0) {
            try {
                ++_waitingReaders;
                wait();
                --_waitingReaders;
            } catch (...) {
                --_waitingReaders;
                throw;
            }
        }
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    short _waitingReaders;
};

```

This implementation uses a member variable `_waitingReaders` to keep track of the number of readers that are suspended. The constructor initializes the variable to zero and the implementation of `get` increments and decrements the variable around the call to `wait`. Note that these statements are enclosed in a `try-catch` block; this ensures that the count of waiting readers remains accurate even if `wait` throws an exception. Finally, `put` calls `notify` only if there is a waiting reader.

The advantage of this implementation is that it minimizes contention on the monitor mutex: a writer wakes up only a single reader at a time, so we do not end up with multiple readers simultaneously trying to lock the mutex. Moreover, the monitor `notify` implementation signals a waiting thread only *after* it has unlocked the mutex. This means that, when a thread wakes up from its call to

`wait` and tries to reacquire the mutex, the mutex is likely to be unlocked. This results in more efficient operation because acquiring an unlocked mutex is typically very efficient, whereas forcefully putting a thread to sleep on a locked mutex is expensive (because it forces a thread context switch).

15.9.4 Efficiency Considerations

The mutual exclusion mechanisms provided by Ice differ in their size and speed. Table 15.1 shows the relative sizes for Windows and Linux (both on Intel architectures).

Table 15.1. Size of mutual exclusion primitives.

Primitive	Size in Bytes (Windows)	Size in Bytes (Linux)
Mutex	24	24
RecMutex	28	28
RWRecMutex	124	60
Monitor<Mutex>	72	40
Monitor<RecMutex>	76	44

Table 15.2 shows the relative performance of the different synchronization primitives under conditions of no lock contention.

Table 15.2. Relative performance of mutual exclusion primitives.

Primitive	Speed (Windows)	Speed (Linux)
Mutex	1.00	1.00
RecMutex	1.02	1.25

Table 15.2. Relative performance of mutual exclusion primitives.

Primitive	Speed (Windows)	Speed (Linux)
RWRecMutex	10.45 (read lock) 22.45 (write lock)	3.23 (read lock) 3.40 (write lock)
Monitor<Mutex>	0.86	1.09
Monitor<RecMutex>	0.90	1.30

Note that the size and performance differences between a mutex and a recursive mutex are negligible, so you should use a non-recursive mutex only if you have a very small critical region that is accessed repeatedly in a tight loop. Similarly, monitors perform much the same as mutexes (but do have a size penalty).

Read-write locks incur a substantial cost in both size and speed (especially under Windows), so you should use a read-write lock only if you can truly exploit the additional parallelism afforded by multiple readers and if a substantial amount of work is performed inside the critical region.

15.10 Threads

As we mentioned in the introduction, the server-side Ice run time creates a thread pool for you and automatically dispatches each incoming request in its own thread. As a result, you usually only need to worry about synchronization among threads to protect critical regions when you implement a server. However, you may wish to create threads of your own. For example, you might need a dedicated thread that responds to input from a user interface. And, if you have complex and long-running operations that can exploit parallelism, you might wish to use multiple threads for the implementation of that operation.

Ice provides a simple thread abstraction that permits you to write portable source code regardless of the native threading platform. This shields you from the native underlying thread APIs and guarantees uniform semantics regardless of your deployment platform.

15.10.1 The Thread Class

The basic thread abstraction in Ice is provided by two classes, `ThreadControl` and `Thread` (defined in `IceUtil/Thread.h`):

```
namespace IceUtil {  
  
    class Time;  
  
    typedef ... ThreadId;           // OS-specific definition  
  
    class ThreadControl {  
    public:  
        ThreadControl();  
  
        ThreadId id() const;  
        void join();  
        void detach();  
        bool isAlive() const;  
        static void sleep(const Time &);  
        static void yield();  
  
        bool operator==(const ThreadControl &) const;  
        bool operator!=(const ThreadControl &) const;  
        bool operator<(const ThreadControl &) const;  
    };  
  
    class Thread {  
    public:  
        ThreadId id() const;  
        virtual void run() = 0;  
        ThreadControl start();  
        ThreadControl getThreadControl() const;  
  
        bool operator==(const Thread &) const;  
        bool operator!=(const Thread &) const;  
        bool operator<(const Thread &) const;  
    };  
    typedef Handle<Thread> ThreadPtr;  
}
```

The `Thread` class is an abstract base class with a pure virtual `run` method. To create a thread, you must specialize the `Thread` class and implement the `run` method (which becomes the starting stack frame for the new thread). The remaining member functions behave as follows:

- `id`

This function returns an identifier unique to each thread, of type `ThreadId`. (`ThreadId` is a platform-specific typedef. Thread IDs are integers.) Thread IDs are useful for debugging or tracing. Calling this method before calling `start` raises a `ThreadNotStartedException`.

- `start`

This member function starts a newly-created thread (that is, it calls the `run` method).

- `getThreadControl`

This member function returns a thread control object for the thread on which it is invoked (see Section 15.10.4). Calling this method before calling `start` raises a `ThreadNotStartedException`.

- `operator==`
`operator !=`
`operator <`

These member functions compare the thread IDs of two threads. They are provided so you can use sorted STL containers with `Thread` objects. Calling these methods before both threads have been started (that is, have valid thread IDs) raises a `ThreadNotStartedException`.

Note that `IceUtil` also defines the type `ThreadPtr`. This is the usual reference-counted smart pointer (see Section 6.14.5) to guarantee automatic clean-up: its destructor calls `delete` to deallocate a dynamically-allocated `Thread` object once its reference count drops to zero.

15.10.2 Implementing Threads

To illustrate how to implement threads, consider the following code fragment:

```
#include <IceUtil/Thread.h>
// ...

Queue q;

class ReaderThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            cout << q.get() << endl;
    }
};
```

```
class WriterThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            q.put(i);
    }
};
```

This code fragment defines two classes, ReaderThread and WriterThread, that inherit from `IceUtil::Thread`. Each class implements the pure virtual `run` method it inherits from its base class. For this simple example, a writer thread places the numbers from 1 to 100 into an instance of the thread-safe `Queue` class we defined in Section 15.9, and a reader thread retrieves 100 numbers from the queue and prints them to `stdout`.

15.10.3 Creating Threads

To create a new thread, we simply instantiate the thread and call its `start` method:

```
IceUtil::ThreadPtr t = new ReaderThread;
t->start();
// ...
```

Note that we assign the return value from `new` to a smart pointer of type `ThreadPtr`. This ensures that we do not suffer a memory leak:

1. When the thread is created, its reference count is set to zero.
2. Prior to calling `run` (which is called by the `start` method), `start` increments the reference count of the thread to 1.
3. For each `ThreadPtr` for the thread, the reference count of the thread is incremented by 1, and for each `ThreadPtr` that is destroyed, the reference count is decremented by 1.
4. When `run` completes, `start` decrements the reference count again and then checks its value: if the value is zero at this point, the `Thread` object deallocates itself by calling `delete`; if the value is non-zero at this point, there are other smart pointers that reference this `Thread` object and deletion happens when the last smart pointer goes out of scope.

Note that, for all this to work, you *must* allocate your `Thread` objects on the heap—stack-allocated `Thread` objects will result in deallocation errors:

```
ReaderThread thread;
IceUtil::ThreadPtr t = &thread; // Bad news!!!
```

This is wrong because the destructor of `t` will eventually call `delete`, which has undefined behavior for a stack-allocated object.

15.10.4 The ThreadControl Class

The `start` method returns an object of type `ThreadControl` (see page 358). The member functions of `ThreadControl` behave as follows:

- `ThreadControl`

The default constructor returns a `ThreadControl` object that refers to the calling thread. This allows you to get a handle to the current (calling) thread even if you do not have saved a handle to that thread previously. For example:

```
IceUtil::ThreadControl self;      // Get handle to self
self.yield();                  // Let another thread run
```

This code fragment causes the calling thread to yield the CPU so some other thread can run. This example also explains why we have two classes, `Thread` and `ThreadControl`: without a separate `ThreadControl`, it would not be possible to obtain a handle to an arbitrary thread. (Note that this code works even if the calling thread was not created by the Ice run time; for example, you can create a `ThreadControl` object for a thread that was created by the operating system.)

- `id`

This function returns an identifier unique to each thread, of type `ThreadId`. (`ThreadId` is a platform-specific typedef. Thread IDs are integers.) Thread IDs are useful for debugging or tracing.

- `join`

This method suspends the calling thread until the thread on which `join` is called has terminated. For example:

```
IceUtil::ThreadPtr t = new ReaderThread; // Create a thread
IceUtil::ThreadControl tc = t->start(); // Start it
```

```
tc.join(); // Wait for it
```

If the reader thread has finished by the time the creating thread calls `join`, the call to `join` returns immediately; otherwise, the creating thread is suspended until the reader thread terminates.

Note that the `join` method of a thread must be called only from one other thread, that is, only one thread can wait for another thread to terminate.

Calling `join` on a thread from more than one other thread has undefined behavior.

Calling `join` on a thread that was previously joined with, or calling `join` on a detached thread has undefined behavior.

- **detach**

This method detaches a thread. Once a thread is detached, it cannot be joined with.

Calling `detach` on an already detached thread, or calling `detach` on a thread that was previously joined with has undefined behavior.

Note that, if you have detached a thread, you must ensure that the detached thread has terminated before your program leaves its `main` function. This means that, because detached threads cannot be joined with, they must have a life time that is shorter than that of the main thread.

- **isAlive**

This method returns true if the underlying thread has not exited yet (that is, has not left its `run` method) and false otherwise. `isAlive` is useful if you want to implement a non-blocking `join`.

- **sleep**

This method suspends a thread for the amount of time specified by the `Time` parameter (see Section 15.8).

- **yield**

This method causes the thread on which `yield` is called to relinquish the CPU, allowing another thread to run.

- **operator==**
operator!=
operator<

As for the `Thread` class, these operators compare thread IDs so you can create sorted STL containers of `ThreadControl` objects.

As for all the synchronization primitives, you must adhere to a few rules when using threads to avoid undefined behavior:

- Do not join with or detach a thread that you have not created yourself.
- For every thread you create, you must either join with that thread exactly once or detach it exactly once; failure to do so may cause resource leaks.
- Do not call `join` on a thread from more than one other thread.
- Do not leave `main` until all other threads you have created have terminated.
- Do not leave `main` until after you have destroyed all `Ice::Communicator` objects you have created (or use the `Ice::Application` class—see Section 10.3.1 on page 237)).
- A common mistake is to call `yield` from within a critical region. Doing so is usually pointless because the call to `yield` will look for another thread that can be run but, when that thread is run, it will most likely try to enter the critical region that is held by the yielding thread and go to sleep again. At best, this achieves nothing and, at worst, it causes many additional context switches for no gain.

If you call `yield`, do so only in circumstances where there is at least a fair chance that another thread will actually be able to run and do something useful.

15.10.5 A Small Example

Following is a small example that uses the `Queue` class we defined in Section 15.9. We create five writer and five reader threads. The writer threads each deposit 100 numbers into the queue, and the reader threads each retrieve 100 numbers and print them to `stdout`:

```
#include <vector>
#include <IceUtil/Thread.h>
// ...

Queue q;

class ReaderThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            cout << q.get() << endl;
    }
};
```

```
class WriterThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            q.put(i);
    }
};

int
main()
{
    vector<IceUtil::ThreadControl> threads;
    int i;

    // Create five reader threads and start them
    //
    for (i = 0; i < 5; ++i) {
        IceUtil::ThreadPtr t = new ReaderThread;
        threads.push_back(t->start());
    }

    // Create five writer threads and start them
    //
    for (i = 0; i < 5; ++i) {
        IceUtil::ThreadPtr t = new WriterThread;
        threads.push_back(t->start());
    }

    // Wait for all threads to finish
    //
    for (vector<IceUtil::ThreadControl>::iterator i
         = threads.begin(); i != threads.end(); ++i) {
        i->join();
    }
}
```

The code uses the `threads` variable, of type `vector<IceUtil::ThreadControl>` to keep track of the created threads. The code creates five reader and five writer threads, storing the `ThreadControl` object for each thread in the `threads` vector. Once all the threads are created and running, the code joins with each thread before returning from `main`.

Note that you *must not* leave `main` without first joining with the threads you have created: many threading libraries crash if you return from `main` with other threads still running. (This is also the reason why you must not terminate a

program without first calling `Communicator::destroy` (see page 236); the `destroy` implementation joins with all outstanding threads before it returns.)

15.11 Portable Signal Handling

The `IceUtil::CtrlCHandler` class provides a portable mechanism to handle Ctrl+C and similar signals sent to a C++ process. On Windows, `IceUtil::CtrlCHandler` is a wrapper for `SetConsoleCtrlHandler`; on POSIX platforms, it handles SIGHUP, SIGTERM and SIGINT with a dedicated thread that waits for these signals using `sigwait`. Signals are handled by a callback function implemented and registered by the user. The callback is a simple function that takes an `int` (the signal number) and returns `void`; it should not throw any exception:

```
namespace IceUtil {  
  
    typedef void (*CtrlCHandlerCallback)(int);  
  
    class CtrlCHandler {  
    public:  
        CtrlCHandler(CtrlCHandlerCallback = 0);  
        ~CtrlCHandler();  
  
        void setCallback(CtrlCHandlerCallback);  
        CtrlCHandlerCallback getCallback() const;  
    };  
}
```

The member functions of `CtrlCHandler` behave as follows:

- constructor

Constructs an instance with a callback function. Only one instance of `CtrlCHandler` can exist in a process at a given moment in time. On POSIX platforms, the constructor masks SIGHUP, SIGTERM and SIGINT, then starts a thread that waits for these signals using `sigwait`. For signal masking to work properly, it is imperative that the `CtrlCHandler` instance be created before starting any thread, and in particular before initializing an Ice communicator.

- **destructor**

Destroys the instance, after which the default signal processing behavior is restored on Windows (`TerminateProcess`). On POSIX platforms, the “sigwait” thread is cancelled and joined, but the signal mask remains unchanged, so subsequent signals are ignored.

- **setCallback**

Sets a new callback function.

- **getCallback**

Gets the current callback function.

It is legal specify a value of zero (0) for the callback function, in which case signals are caught and ignored until a non-zero callback function is set.

A typical use for `CtrlCHandler` is to shutdown a communicator in an Ice server (see Section 10.3.1).

15.12 Summary

This chapter explained the threading abstractions provided by Ice: mutexes, monitors, and threads. Using these APIs allows to make your code thread safe and to create threads of your own without having to use non-portable APIs that differ in syntax or semantics across different platforms: Ice not only provides a portable API but also guarantees that the semantics of the various functions are the same across different platforms. This makes it easier to create thread-safe applications and allows you to move your code between platforms with simple recompilation.

Part III

Advanced Ice



Chapter 16

The Ice Run Time in Detail

16.1 Introduction

Now that we have seen the basics of implementing clients and servers, it is time to look at the Ice run time in more detail. This chapter presents the server-side APIs of the Ice run time in detail for synchronous, oneway, and datagram invocations in detail. (We cover the asynchronous interfaces in Chapter 17.)

Section 16.2 describes the functionality associated with Ice communicators, which are the main handle to the Ice run time. Sections 16.3 to 16.5 describe object adapters and the role they play for call dispatch, and show the relationship between proxies, Ice objects, servants, and object identities. Section 16.6 describes servant locators, which are a major mechanism in Ice for controlling the trade-off between performance and memory consumption. Section 16.7 describes the most common implementation techniques that are used by servers. We suggest that you read this section in detail because knowledge of these techniques is crucial to building systems that perform and scale well. Section 16.8 describes implicit transmission of parameters from client to server and Section 16.9 discusses invocation timeouts. Sections 16.10 to 16.13 describe oneway, datagram, and batched invocations, and Sections 16.14 to 16.16 deal with logging, statistics collection, and location transparency. Finally, Section 16.17 compares the Ice server-side run time with the corresponding CORBA approach.

16.2 Communicators

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. An instance of `Ice::Communicator` is associated with a number of run-time resources:

- client-side thread pool

The client-side thread pool ensures that at least one thread is available on the client side to receive replies for outstanding requests. This ensures that no deadlock can occur. For example, if a server calls back into the client from within an operation implementation, the client-side receiver thread can process the request from the server even though the client is waiting for a reply for its request from the same server.

The client-side thread pool is also used for asynchronous method invocation (AMI), to avoid deadlocks in callbacks (see Chapter 17).

- server-side thread pool

Threads in this pool accept incoming connections and handle requests from clients.

- configuration properties

Various aspects of the Ice run time can be configured via properties. Each communicator has its own set of such configuration properties (see Chapter 14).

- object factories

In order to instantiate classes that are derived from a known base type, the communicator maintains a set of object factories that can instantiate the class on behalf of the Ice run time (see Section 6.14.4 and Section 8.14.1).

- a logger object

A logger object implements the `Ice::Logger` interface and determines how log messages that are produced by the Ice run time are handled (see Section 16.14).

- a statistics object

A statistics object implements the `Ice::Stats` interface and is informed about the amount of traffic (bytes sent and received) that is handled by a communicator (see Section 16.15).

- a default router

A router implements the `Ice::Router` interface. Routers are used by Glacier (see Chapter 25) to implement the firewall functionality of Ice.

- a default locator

A locator is an object that resolves an object identity to a proxy. Locator objects are used to build location services, such as IcePack (see Chapter 21).

- a plug-in manager

Plug-ins are objects that add features to a communicator. For example, IceSSL (see Chapter 24) is implemented as a plug-in. Each communicator has a plug-in manager that implements the `Ice::PluginManager` interface and provides access to the set of plug-ins for a communicator.

- object adapters

Object adapters dispatch incoming requests and take care of passing each request to the correct servant.

Object adapters and objects that use different communicators are completely independent from each other. Specifically:

- Each communicator uses its own thread pool. This means that if, for example, one communicator runs out of threads for incoming requests, only objects using that communicator are affected. Objects using other communicators have their own thread pool and are therefore unaffected.
- Collocated invocations across different communicators are not optimized, whereas collocated invocations using the same communicator bypass much of the overhead of call dispatch.

Typically, servers use only a single communicator but, occasionally, multiple communicators can be useful. For example, IceBox (see Chapter 26) uses a separate communicator for each Ice service it loads to ensure that different services cannot interfere with each other. Multiple communicators are also useful to avoid thread starvation: if one service runs out of threads, this leaves the remaining services unaffected.

The interface of the communicator is defined in Slice. Part of this interface looks as follows:

```
module Ice {
    local interface Communicator {
        string proxyToString(Object* obj);
        Object* stringToProxy(string str);
        ObjectAdapter createObjectAdapter(string name);
```

```
ObjectAdapter createObjectAdapterWithEndpoints(
    string name,
    string endpoints);
void shutdown();
void waitForShutdown();
void destroy();
// ...
};

// ...
};
```

The communicator offers a number of operations:

- `proxyToString`
- `stringToProxy`

These operations allow you to convert a proxy into its stringified representation and vice versa.

- `createObjectAdapter`
- `createObjectAdapterWithEndpoints`

These operations create a new object adapter. Each object adapter is associated with one or more transport endpoints. Typically, an object adapter has a single transport endpoint. However, an object adapter can also offer multiple endpoints. If so, these endpoints each lead to the same set of objects and represent alternative means of accessing these objects. This is useful, for example, if a server is behind a firewall but must offer access to its objects to both internal and external clients; by binding the adapter to both the internal and external interfaces, the objects implemented in the server can be accessed via either interface.

Whereas `createObjectAdapter` determines which endpoints to bind itself to from configuration information (see Chapter 21), `createObjectAdapterWithEndpoints` allows you to specify the transport endpoints for the new adapter. Typically, you should use `createObjectAdapter` in preference to `createObjectAdapterWithEndpoints`. Doing so keeps transport-specific information, such as host names and port numbers, out of the source code and allows you to reconfigure the application by changing a property (and so avoid recompilation when a transport endpoint needs to be changed).

- `shutdown`

This operation shuts down the server side of the Ice run time:

- Operation invocations that are in progress at the time `shutdown` is called are allowed to complete normally. `shutdown` does *not* wait for these operations to complete; when `shutdown` returns, you know that no new incoming requests will be dispatched, but operations that were already in progress at the time you called `shutdown` may still be running. You can wait for still executing operations to complete by calling `waitForShutdown`.
- Operation invocations that arrive after the server has called `shutdown` either fail with a `ConnectFailedException` or are transparently redirected to a new instance of the server (see Chapter 21).

- **`waitForShutdown`**

This operation suspends the calling thread until the communicator has shutdown (that is, until no more operations are executing in the server). This allows you to wait until the server is idle before you destroy the communicator.

- **`destroy`**

This operation destroys the communicator and all its associated resources, such as threads, communication endpoints, and memory resources. Once you have destroyed the communicator (and therefore destroyed the run time for that communicator), you must not call any other Ice operation (other than to create another communicator).

It is imperative that you call `destroy` before you leave the `main` function of your program. Failure to do so results in undefined behavior.

Calling `destroy` before leaving `main` is necessary because `destroy` waits for all running threads to terminate before it returns. If you leave `main` without calling `destroy`, you will leave `main` with other threads still running; many threading packages do not allow you to do this and end up crashing your program.

If you call `destroy` without calling `shutdown`, the call waits for all executing operation invocations to complete before it returns (that is, the implementation of `destroy` implicitly calls `shutdown` followed by `waitForShutdown`). `shutdown` (and, therefore, `destroy`) deactivate all object adapters that are associated with the communicator.

One the client side, calling `shutdown` while operations are still executing causes those operations to terminate with a `CommunicatorDestroyedException`.

16.3 Object Adapters

A communicator contains one or more object adapters. An object adapter sits at the boundary between the Ice run time and the server application code and has a number of responsibilities:

- It maps Ice objects to servants for incoming requests and dispatches the requests to the application code in each servant (that is, an object adapter implements an up-call interface that connects the Ice run time and the application code in the server).
- It assists in life cycle operations so Ice objects and servants can be created and existing destroyed without race conditions.
- It provides one or more transport endpoints. Clients access the Ice objects provided by the adapter via those endpoints. (It is also possible to create an object adapter without endpoints. In this case the adapter is used for bidirectional callbacks—see Chapter 25.)

Each object adapter has one or more servants that incarnate Ice objects, as well as one or more transport endpoints. If an object adapter has more than one endpoint, all servants registered with that adapter respond to incoming requests on any of the endpoints. In other words, if an object adapter has multiple transport endpoints, those endpoints represent alternative communication paths to the same set of objects (for example, via different transports).

Each object adapter belongs to exactly one communicator (but a single communicator can have many object adapters).

Each object adapter can optionally have its own thread pool, enabled via the `<adapter-name>.ThreadPool.Size` property (see Appendix C). If so, client invocations for that adapter are dispatched in a thread taken from the adapter's thread pool instead of using a thread from the communicator's server thread pool.

16.3.1 The Active Servant Map

Each object adapter maintains a data structure known as the *active servant map*. The active servant map (or *ASM*, for short) is a lookup table that maps object identities to servants: for C++, the lookup value is a smart pointer to the corresponding servant's location in memory; for Java, the lookup value is a Java reference to the servant. When a client sends an operation invocation to the server, the request is targeted at a specific transport endpoint. Implicitly, the transport endpoint identi-

fies the object adapter that is the target of the request (because no two object adapters can be bound to the same endpoint). The proxy via which the client sends its request contains the object identity for the corresponding object, and the client-side run time sends this object identity over the wire with the invocation. In turn, the object adapter uses that object identity to look in its ASM for the correct servant to dispatch the call to, as shown in Figure 16.1.

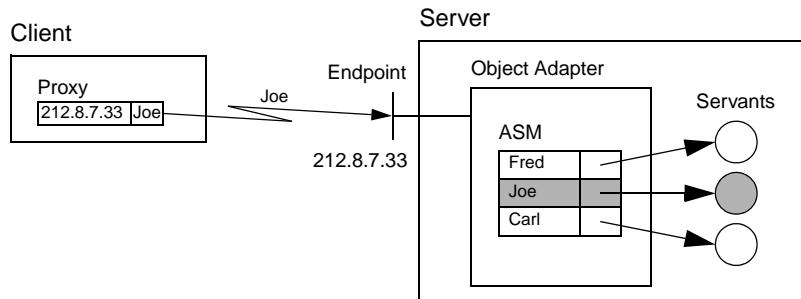


Figure 16.1. Binding a request to the correct servant.

The process of associating a request via a proxy to the correct servant is known as *binding*. The scenario depicted in Figure 16.1 shows direct binding, in which the transport endpoint is embedded in the proxy. Ice also supports an indirect binding mode, in which the correct transport endpoints are provided by the IcePack service (see Chapter 21 for details).

If a client request contains an object identity for which there is no entry in the adapter's ASM, the adapter returns an `ObjectNotExistException` to the client (unless you use a servant locator—see Section 16.6).

16.3.2 Servants

As mentioned in Section 2.2.2, servants are the physical manifestation of an Ice object, that is, they are entities that are implemented in a concrete programming language and instantiated in the server's address space. Servants provide the server-side behavior for operation invocations sent by clients.

The same servant can be registered with one or more object adapters.

16.3.3 Object Adapter Interface

Object adapters are local interfaces:

```
module Ice {  
    local interface ObjectAdapter {  
        string getName();  
  
        Communicator getCommunicator();  
  
        // ...  
    };  
};
```

The operations behave as follows:

- The `getName` operation returns the name of the adapter as passed to `Communicator::createObjectAdapter` or `Communicator::createObjectAdapterWithEndpoints`.
- The `getCommunicator` operation returns the communicator that was used to create the adapter.

Note that there are other operations in the `ObjectAdapter` interface; we will explore these throughout the remainder of this chapter.

16.3.4 Servant Activation and Deactivation

The term *servant activation* refers to making the presence of a servant for a particular Ice object known to the Ice run time. Activating a servant adds an entry to the active servant map shown in Figure 16.1. Another way of looking at servant activation is to think of it as creating a link between the identity of an Ice object and the corresponding programming-language servant that handles requests for that Ice object. Once the Ice run time has knowledge of this link, it can dispatch incoming requests to the correct servant. Without this link, that is, without a corresponding entry in the ASM, an incoming request for the identity results in an `ObjectNotExistException`. While a servant is activated, it is said to *incarnate* the corresponding Ice object.

The inverse operation is known as *servant deactivation*. Deactivating a servant removes an entry for a particular identity from the ASM. Thereafter, incoming requests for that identity are no longer dispatched to the servant and result in an `ObjectNotExistException`.

The object adapter offers a number of operations to manage servant activation and deactivation:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        Object* add(Object servant, Identity id);
        Object* addWithUUID(Object servant);
        Object remove(Identity id);

        // ...
    };
};
```

The operations behave as follows:

- **add**

The `add` operation adds a servant with the given identity to the ASM. Requests are dispatched to that servant as soon as `add` is called. The return value is the proxy for the Ice object incarnated by that servant. The proxy embeds the identity passed to `add`.

You cannot call `add` with the same identity more than once: attempts to add an already existing identity to the ASM result in an `AlreadyRegisteredException`. (It does not make sense to add two servants with the same identity because that would make it ambiguous as to which servant should handle incoming requests for that identity.)

Note that it is possible to activate the same servant multiple times with different identities. In that case, the same single servant incarnates multiple Ice objects. We explore the ramifications of this in more detail in Section 16.7.2.

- **addWithUUID**

The `addWithUUID` operation behaves the same way as the `add` operation but does not require you to supply an identity for the servant. Instead, `addWithUUID` generates a UUID (see [14]) as the identity for the corresponding Ice object. You can retrieve the generated identity by calling the `ice_getIdentity` operation on the returned proxy. `addWithUUID` is useful to create identities for temporary objects, such as short-lived session objects. (You can also use `addWithUUID` for persistent objects that do not have a natural identity, as we have done for the filesystem application.)

- **remove**

The `remove` operation breaks the association between an identity and its servant by removing the corresponding entry from the ASM; it returns a smart pointer to the removed servant.

Once the servant is deactivated, new incoming requests for the removed identity result in an `ObjectNotExistException`. Requests that are executing inside the servant at the time `remove` is called are allowed to complete normally. Once the last request for the servant is complete, the object adapter drops its reference (or smart pointer, for C++) to the servant. At that point, the servant becomes available for garbage collection (or is destroyed, for C++), provided that you do not hold references or smart pointers to the servant elsewhere. The net effect is that a deactivated servant is destroyed once it becomes idle.

Deactivating an object adapter (see Section 16.3.5) implicitly calls `remove` on its active servants.

16.3.5 Adapter States

An object adapter has a number of processing states:

- **holding**

In this state, any incoming requests for the adapter are held, that is, not dispatched to servants.

For TCP/IP (and other stream-oriented protocols), the server-side run time stops reading from the corresponding transport endpoint while the adapter is in the holding state. In addition, it also does not accept incoming connection requests from clients. This means that if a client sends a request to an adapter that is in the holding state, the client eventually receives a `TimeoutException` or `ConnectTimeoutException` (unless the adapter is placed into the active state before the timer expires).

For UDP, client requests that arrive at an adapter that is in the holding state are thrown away.

Immediately after creation of an adapter, the adapter is in the holding state. This means that requests are not dispatched until you place the adapter into the active state.

- active

In this state, the adapter accepts incoming requests and dispatches them to servants. A newly-created adapter is initially in the holding state. The adapter begins dispatching requests as soon as you place it into the active state.

You can transition between the active and the holding state as many times as you wish.

- inactive

In this state, the adapter has conceptually been destroyed (or is in the process of being destroyed). Deactivating an adapter destroys all transport endpoints that are associated with the adapter. Requests that are executing at the time the adapter is placed into the inactive state are allowed to complete, but no new requests are accepted. (New requests are rejected with an exception). Once an adapter has been deactivated, you cannot reactivate it again in the same process.

The `ObjectAdapter` interface offers operations that allow you to change the adapter state, as well as to wait for a state change to be complete:

```
module Ice {  
    local interface ObjectAdapter {  
        // ...  
  
        void activate();  
        void hold();  
        void waitForHold();  
        void deactivate();  
        void waitForDeactivate();  
  
        // ...  
    };  
};
```

The operations behave as follows:

- **activate**

The `activate` operation places the adapter into the active state. Activating an adapter that is already active has no effect. The Ice run time starts dispatching requests to servants for the adapter as soon as `activate` is called.

- **hold**

The `hold` operation places the adapter into the holding state. Requests that arrive after calling `hold` are held as detailed on page 378. Requests that are in

progress at the time `hold` is called are allowed to complete normally. Note that `hold` returns immediately without waiting for currently executing requests to complete.

- **`waitForHold`**

The `waitForHold` operation suspends the calling thread until the adapter has completed its transition to the holding state, that is, until all currently executing requests have finished. You can call `waitForHold` from multiple threads, and you can call `waitForHold` while the adapter is in the active state. Calling `waitForHold` on an adapter in the inactive state does nothing and returns immediately.

- **`deactivate`**

The `deactivate` operation places the adapter into the inactive state. Requests that arrive after calling `deactivate` are rejected, but currently executing requests are allowed to complete. Once an adapter is in the inactive state, it cannot be reactivated again. Calling `activate`, `hold`, `waitForHold`, or `deactivate` on an adapter that is in the inactive state has no effect. Any servants associated with the adapter are destroyed once they become idle. Note that `deactivate` returns immediately without waiting for the currently executing requests to complete.

- **`waitForDeactivate`**

The `waitForDeactivate` operation suspends the calling thread until the adapter has completed its transition to the inactive state, that is, until all currently executing requests have completed, all transport endpoints have been closed, and all associated servants have been destroyed. You can call `waitForDeactivate` from multiple threads, and you can call `waitForDeactivate` while the adapter is in the active or holding state. Calling `waitForDeactivate` on an adapter that is in the inactive state does nothing and returns immediately.

Placing an adapter into the holding state is useful, for example, if you need to make state changes in the server that require the server (or a group of servants) to be idle. For example, you could place the implementation of your servants into a dynamic library and upgrade the implementation by loading a newer version of the library at run time without having to shut down the server.

Similarly, waiting for an adapter to complete its transition to the inactive state is useful if your server needs to perform some final clean-up work that cannot be carried out until all executing request have completed.

16.4 Object Identity

Each Ice object has an object identity defined as follows:

```
module Ice {  
    struct Identity {  
        string name;  
        string category;  
    };  
};
```

As you can see, an object identity consists of a pair of strings, a `name` and a `category`. Either the `name` or the `category` can be empty. (If a proxy contains an identity in which both `name` and `category` are empty, Ice interprets that proxy as a null proxy.) The complete object identity is the combination of `name` and `category`, that is, for two identities to be equal, both `name` and `category` must be the same. The `category` member is usually the empty string, unless you are using servant locators (see Section 16.6).¹

Object identities can be represented as strings; the `category` part appears first and is followed by the `name`; the two components are separated by a / character, for example:

`Factory/File`

In this example, `Factory` is the `category`, and `File` is the `name`. If the `name` or `category` member themselves contain a / character, the stringified representation escapes the / character with a \, for example:

`Factories\Factory\Node\\File`

In this example, the `category` member is `Factories/Factory` and the `name` member is `Node/File`.

There are quite a number of other characters that must be escaped in stringified identities. To make conversion of identities to and from strings easier, the Ice run time provides utility functions that encode and decode identities.

For C++, the utility functions are defined as follows:

1. Glacier (see Chapter 25) also uses the `category` member for filtering.

```
namespace Ice {
    std::string identityToString(const Ice::Identity id);
    Ice::Identity stringToIdentity(const std::string& s);
};
```

For Java, the utility functions are in the `Ice.Util` class and are defined as:

```
package Ice;

public final class Util {
    public static String identityToString(Identity id);
    public static Identity stringToIdentity(String s);
}
```

These functions correctly encode and decode characters that might otherwise cause problems (such as control characters or white space).

As mentioned on page 377, each entry in the ASM for an object adapter must be unique: you cannot add two servants with the same identity to the ASM.

16.5 The `Ice::Current` Object

Up to now, we have tacitly ignored the trailing parameter of type `Ice::Current` that is passed to each skeleton operation on the server side. The `Current` object is defined as follows:

```
module Ice {
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Nonmutating, \Idempotent };

    local struct Current {
        ObjectAdapter    adapter;
        Identity        id;
        string          facet;
        string          operation;
        OperationMode   mode;
        Context         ctx;
    };
};
```

Note that the `Current` object provides access to information about the currently executing request to the implementation of an operation in the server:

- **adapter**

The `adapter` member provides access to the object adapter via which the request is being dispatched. In turn, the adapter provides access to its communicator (via the `getCommunicator` operation).

- **id**

The `id` member provides the object identity for the current request (see Section 16.4).

- **facet**

The `facet` member provides access to the facet for the request (see Chapter 18).

- **operation**

The `operation` member contains the name of the operation that is being invoked. Note that the operation name may indicate one of the operations on `Ice::Object`, such as `ice_ping` or `ice_isA`.

- **mode**

The `mode` member contains the invocation mode for the operation (`Normal`, `Idempotent`, or `Nonmutating`).

- **ctx**

The `ctx` member contains the current context for the invocation (see Section 16.8).

If you implement your server such that it uses a separate servant for each Ice object, the contents of the `Current` object are not particularly interesting. (You would most likely access the `Current` object to read the `adapter` member, for example, to activate or deactivate a servant.) However, as we will see in Section 16.6, the `Current` object is essential for more sophisticated (and more scalable) servant implementations.

16.6 Servant Locators

Using an adapter's ASM to map Ice objects to servants has a number of design implications:

- Each Ice object is represented by a different servant.²
- All servants for all Ice objects are permanently in memory.

Using a separate servant for each Ice object in this fashion is common to many server implementations: the technique is simple to implement and provides a natural mapping from Ice objects to servants. Typically, on start-up, the server instantiates a separate servant for each Ice object, activates each servant, and then calls `activate` on the object adapter to start the flow of requests.

There is nothing wrong with the above design, provided that two criteria are met:

1. The server has sufficient memory available to keep a separate servant instantiated for each Ice object at all times.
2. The time required to initialize all the servants on start-up is acceptable.

For many servers, neither criterion presents a problem: provided that the number of servants is small enough and that the servants can be initialized quickly, this is a perfectly acceptable design. However, the design does not scale well: the memory requirements of the server grow linearly with the number of Ice objects so, if the number of objects gets too large (or if each servant stores too much state), the server runs out of memory.

Ice uses *servant locators* to allow you to scale servers to larger numbers of objects.

16.6.1 Overview

In a nutshell, a servant locator is a local object that you implement and attach to an object adapter. Once an adapter has a servant locator, it consults its ASM to locate a servant for an incoming request as usual. If a servant for the request can be found in the ASM, the request is dispatched to that servant. However, if the ASM does not have an entry for the object identity of the request, the object adapter calls back into the servant locator to ask it to provide a servant for the request. The servant locator either

- instantiates a servant and passes it to the Ice run time, in which case the request is dispatched to that newly instantiated servant, or
- the servant locator indicates failure to locate a servant to the Ice run time, in which case the client receives an `ObjectNotExistException`.

2. It is possible to register a single servant with multiple identities. However, there is little point in doing so because a default servant (see Section 16.7.2) achieves the same thing.

This simple mechanism allows us to scale servers to provide access to an unlimited number of Ice objects: instead of instantiating a separate servant for each and every Ice object in existence, the server can instantiate servants for only a subset of Ice objects, namely those that are actually used by clients.

Servant locators are most commonly used by servers that provide access to databases: typically, the number of entries in the database is far larger than what the server can hold in memory. Servant locators allow the server to only instantiate servants for those Ice objects that are actually used by clients.

Another common use for servant locators is in servers that are used for process control or network management: in that case, there is no database but, instead, there is a potentially very large number of devices or network elements that must be controlled via the server. Otherwise, this scenario is the same as for large databases: the number of Ice objects exceeds the number of servants that the server can hold in memory and, therefore, requires an approach that allows the number of instantiated servants to be less than the number of Ice objects.

16.6.2 Servant Locator Interface

A servant locator has the following interface:

```
module Ice {
    local interface ServantLocator {
        Object locate(    Current      curr,
                          out LocalObject cookie);

        void finished(   Current      curr,
                         Object       servant,
                         LocalObject cookie);

        void deactivate(string category);
    };
};
```

Note that `ServantLocator` is a local interface. To create an actual implementation of a servant locator, you must define a class that is derived from `Ice::ServantLocator` and provide implementations of the `locate`, `finished`, and `deactivate` operations. The Ice run time invokes the operations on your derived class as follows:

- `locate`

Whenever a request arrives for which no entry exists in the ASM, the Ice run time calls `locate`. The implementation of `locate` (which you provide as part

of the derived class) is supposed to return a servant that can process the incoming request. Your implementation of `locate` can behave in three possible ways:

1. Instantiate and return a servant for the current request. In this case, the Ice run time dispatches the request to the newly instantiated servant.
2. Return null. In this case, the Ice run time raises an `ObjectNotExistException` in the client.
3. Throw an exception. In this case, the Ice run time propagates the thrown exception back to the client. (Keep in mind that all exceptions, apart from `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`, are presented as `UnknownLocalException` to the client.)

The `cookie` out-parameter to `locate` allows you return a local object to the object adapter. The object adapter does not care about the contents of that object (and it is legal to return a null cookie). Instead, the Ice run time passes whatever cookie you return from `locate` back to you when it calls `finished`. This allows you to pass an arbitrary amount of state from `locate` to the corresponding call to `finished`.

- **finished**

If a call to `locate` has returned a servant to the Ice run time, the Ice run time dispatches the incoming request to the servant. Once the request is complete (that is, the operation being invoked has completed), the Ice run time calls `finished`, passing the servant whose operation has completed, the `Current` object for the request, and the cookie that was initially created by `locate`. This means that every call to `locate` is balanced by a corresponding call to `finished` (provided that `locate` actually returned a servant).

You cannot throw exceptions from `finished`—if you do, the Ice run time logs an error messages and (for connection-oriented transports) closes the connection to the client.

- **deactivate**

The `deactivate` operation is called by the Ice run time when the object adapter to which the servant locator is attached is deactivated. This allows the servant locator to clean up (for example, the locator might close a database connection). The Ice run time passes the category of the servant locator being deactivated to the `deactivate` operation.

It is important to realize that the Ice run time does not “remember” the servant that is returned by a particular call to `locate`. Instead, the Ice run time simply dispatches an incoming request to the servant returned by `locate` and, once the request is complete, calls `finished`. In particular, if two requests for the same servant arrive more or less simultaneously, the Ice run time calls `locate` and `finished` once for each request. In other words, `locate` establishes the association between an object identity and a servant; that association is valid only for a single request and is never used by the Ice run time to dispatch a different request.

16.6.3 Threading Guarantees for Servant Locators

The Ice run time guarantees that every operation invocation that involves a servant locator is bracketed by calls to `locate` and `finished`, that is, every call to `locate` is balanced by a corresponding call to `finished` (assuming that the call to `locate` actually returned a servant, of course).

In addition, the Ice run time guarantees that `locate`, the operation, and `finished` are called by the same thread. This guarantee is important because it allows you to use `locate` and `finished` to implement thread-specific pre- and post-processing around operation invocations. (For example, you can start a transaction in `locate` and commit or rollback that transaction in `finished`, or you can acquire a lock in `locate` and release the lock in `finished`.³)

Note that, if you are using asynchronous method dispatch (see Chapter 17), the thread that starts a call is not necessarily the thread that finishes it. In that case, `finished` is called by whatever thread executes the operation implementation, which is may be different thread than the one that called `locate`.

The Ice run time also guarantees that `deactivate` is called when you deactivate the object adapter to which the servant locator is attached. The `deactivate` call is made only once all operations that involved the servant locator are finished, that is, `deactivate` is guaranteed not to run concurrently with `locate` or `finished`, and is guaranteed to be the *last* call made to a servant locator.

Beyond this, the Ice run time provides no threading guarantees for servant locators. In particular:

- It is possible for invocations of `locate` to proceed concurrently (for the same object identity or for different object identities).

3. Both transactions and locks usually are thread-specific, that is, only the thread that started a transaction can commit it or roll it back, and only the thread that acquired a lock can release the lock.

- It is possible for invocations of `finished` to proceed concurrently (for the same object identity or for different object identities).
- It is possible for invocations of `locate` and `finished` to proceed concurrently (for the same object identity or for different object identities).

These semantics allow you to extract the maximum amount of parallelism from your application code (because the Ice run time does not serialize invocations when serialization may not be necessary). Of course, this means that you must protect access to shared data from `locate` and `finished` with mutual exclusion primitives as necessary.

16.6.4 Servant Locator Registration

An object adapter does not automatically know when you create a servant locator. Instead, you must explicitly register servant locators with the object adapter:

```
module Ice {  
    local interface ObjectAdapter {  
        // ...  
  
        void addServantLocator(ServantLocator locator,  
                               string          category);  
  
        ServantLocator findServantLocator(string category);  
  
        // ...  
    };  
};
```

As you can see, the object adapter allows you add and find servant locators. Note that, when you register a servant locator, you must provide an argument for the `category` parameter. The value of the `category` parameter controls which object identities the servant locator is responsible for: only object identities with a matching `category` member (see page 381) trigger a corresponding call to `Locate`. An incoming request for which no explicit entry exists in the ASM and with a category for which no servant locator is registered returns an `ObjectNotExistException` to the client.

`addServantLocator` has the following semantics:

- You can register exactly one servant locator for a specific category. Attempts to call `addServantLocator` for the same category more than once raise an `AlreadyRegisteredException`.

- You can register different servant locators for different categories, or you can register the same single servant locator multiple times (each time for a different category). In the former case, the category is implicit in the servant locator instance that is called by the Ice run time; in the latter case, the implementation of `locate` can find out which category the incoming request is for by examining the `object identity` member of the `Current` object that is passed to `locate`.
- It is legal to register a servant locator for the empty category. Such a servant locator is known as a *default servant locator*: if a request comes in for which no entry exists in the ASM, and whose category does not match the category of any other registered servant locator, the Ice run time calls `locate` on the default servant locator.

Note that, once registered, you cannot change or remove the servant locator for a category. The life cycle of the servant locators for an object adapter ends with the life cycle of the adapter: when the object adapter is deactivated, so are its servant locators.

The `findServantLocator` operation allows you retrieve the servant locator for a specific category (including the empty category). If no servant locator is registered for the specified category, `findServantLocator` returns null.

Call Dispatch Semantics

The preceding rules may seem complicated, so here is a summary of the actions taken by the Ice run time to locate a servant for an incoming request.

Every incoming request implicitly identifies a specific object adapter for the request (because the request arrives at a specific transport endpoint and, therefore, identifies a particular object adapter). The incoming request carries an object identity that must be mapped to a servant. To locate a servant, the Ice run time goes through the following steps, in the order shown:

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request to the corresponding servant.
2. If the category of the incoming object identity is non-empty, look for a servant locator that is registered for that category. If such a servant locator is registered, call `locate` on the servant locator and, if `locate` returns a servant, dispatch the request to that servant, followed by a call to `finished`; otherwise, if the call to `locate` returns null, raise `ObjectNotExistException` in the client.

3. If the category of the incoming object identity is empty, or no servant locator could be found for the category in step 2, look for a default servant locator (that is, a servant locator that is registered for the empty category). If a default servant locator is registered, dispatch the request as for step 2.
4. Raise `ObjectNotExistException` in the client.

It is important to keep these call dispatch semantics in mind because they enable a number of powerful implementation techniques. Each technique allows you to streamline your server implementation and to precisely control the trade-off between performance, memory consumption, and scalability. To illustrate the possibilities, we outline a number of the most common implementation techniques in the following section.

16.6.5 Implementing a Simple Servant Locator

To illustrate the concepts outlined in the previous sections, let us examine a (very simple) implementation of a servant locator. Consider that we want to create an electronic phone book for the entire world's telephone system (which, clearly, involves a very large number of entries, certainly too many to hold the entire phone book in memory). The actual phone book entries are kept in a large database. Also assume that we have a search operation that returns the details of a phone book entry. The Slice definitions for this application might look something like the following:

```
struct Details {
    // Lots of details about the entry here...
};

interface PhoneEntry {
    nonmutating Details getDetails();
    idempotent void updateDetails(Details d);
    // ...
};

struct SearchCriteria {
    // Fields to permit searching...
};

interface PhoneBook {
    nonmutating PhoneEntry* search(SearchCriteria c);
    // ...
};
```

The details of the application do not really matter here; the important point to note is that each phone book entry is represented as an interface for which we need to create a servant eventually, but we cannot afford to keep servants for all entries permanently in memory.

Each entry in the phone database has a unique identifier. This identifier might be an internal database identifier, or a combination of field values, depending on exactly how the database is constructed. The important point is that we can use this database identifier to link the proxy for an Ice object to its persistent state: we simply use the database identifier as the object identity. This means that each proxy contains the primary access key that is required to locate the persistent state of each Ice object and, therefore, instantiate a servant for that Ice object.

What follows is an outline implementation in C++. The class definition of our servant locator looks as follows:

```
class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr & cookie);

    virtual void finished(const Ice::Current & c,
                         const Ice::ObjectPtr & servant,
                         const Ice::LocalObjectPtr & cookie);

    virtual void deactivate(const std::string & category);
};
```

Note that `MyServantLocator` inherits from `Ice::ServantLocator` and implements the pure virtual functions that are generated by the `slice2cpp` compiler for the `Ice::ServantLocator` interface. Of course, as always, you can add additional member functions, such as a constructor and destructor, and you can add private data members as necessary to support your implementation.

In C++, you can implement the `locate` member function along the following lines:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current & c,
                        Ice::LocalObjectPtr & cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;
```

```
// Use the identity to retrieve the state from the database.  
//  
ServantDetails d;  
try {  
    d = DB_lookup(name);  
} catch (const DB_error &)  
    throw Ice::ObjectNotExistException(__FILE__, __LINE__);  
  
// We have the state, instantiate a servant and return it.  
//  
return new PhoneEntryI(d);  
}
```

For the time being, the implementations of `finished` and `deactivate` are empty and do nothing.

The `DB_lookup` call in the preceding example is assumed to access the database. If the lookup fails (presumably, because no matching record could be found), `DB_lookup` throws a `DB_error` exception. The code catches that exception and throws an `Ice::ObjectNotExistException` in its stead; that exception is returned to the client to indicate that the client used a proxy to a no-longer existent Ice object.

Note that `locate` instantiates the servant on the heap and returns it to the Ice run time. This raises the question of when the servant will be destroyed. The answer is that the Ice run time holds onto the servant for as long as necessary, that is, long enough to invoke the operation on the returned servant and to call `finished` once the operation has completed. Thereafter, the servant is no longer needed and the Ice run time destroys the smart pointer that was returned by `locate`. In turn, because no other smart pointers exist for the same servant, this causes the destructor of the `PhoneEntryI` instance to be called, and the servant to be destroyed.

The upshot of this design is that, for every incoming request, we instantiate a servant and allow the Ice run time to destroy the servant once the request is complete. Depending on your application, this may be exactly what is needed, or it may be prohibitively expensive—we will explore designs that avoid creation and destruction of a servant for every request shortly.

In Java, the implementation of our servant locator looks very similar:

```
public class MyServantLocator
    extends Ice.LocalObjectImpl
    implements Ice.ServantLocator {

    public Ice.Object locate(Ice.Current c,
                            Ice.LocalObjectHolder cookie)
    {
        // Get the object identity. (We use the name member
        // as the database key.
        String name = c.id.name;

        // Use the identity to retrieve the state
        // from the database.
        //
        ServantDetails d;
        try {
            d = DB.lookup(name);
        } catch (DB.error e) {
            throw new Ice.ObjectNotExistException();
        }

        // We have the state, instantiate a servant and return it.
        //
        return new PhoneEntryI(d);
    }

    public void finished(Ice.Current c,
                        Ice.Object servant,
                        Ice.LocalObject cookie)
    {
    }

    public void deactivate(String category)
    {
    }
}
```

Note that the `MyServantLocator` class extends `Ice.LocalObjectImpl`. `LocalObjectImpl` is provided by the Ice run time and contains the implementation of methods that are common to all local objects (namely, `equals`, `clone`, and `ice_hash`).

A word of warning about the Java mapping: if you want to keep a copy of one of the members of `Ice.Current`, or a copy of `Ice.Current` itself beyond the scope of `locate` or `finished`, you *must* make a copy of the value instead of just storing a reference: for efficiency reasons, the Java Ice run time does not

allocate separate objects for each member of a `Current` object. Instead, a pool of `Current` objects is reused for different invocations and the members of the various `Current` objects are assigned to just prior to call dispatch. This means that, if you store a reference to, for example, the identity inside a `Current` object, that identity may unexpectedly change later when the Ice run-time reuses that `Current` object.⁴

All implementations of `locate` follow the pattern illustrated by the previous pseudo-code:

1. Use the `id` member of the passed `Current` object to obtain the object identity. Typically, only the `name` member of the identity is used to retrieve servant state. The `category` member is normally used to select a servant locator. (We will explore use of the `category` member shortly.)
2. Retrieve the state of the Ice object from secondary storage (or the network) using the object identity as a key.
 - If the lookup succeeds, you have retrieved the state of the Ice object.
 - If the lookup fails, throw an `ObjectNotExistException`. In that case, the Ice object for the client's request truly does not exist, presumably, because that Ice object was deleted earlier, but the client still has a proxy to the now-deleted object.
3. Instantiate a servant and use the state retrieved from the database to initialize the servant. (In this example, we pass the retrieved state to the servant constructor.)
4. Return the servant.

Of course, before we can use our servant locator, we must inform the adapter of its existence prior to activating the adapter, for example (in Java):

```
MyServantLocator sl = new MyServantLocator();  
adapter.addServantLocator(sl, "");
```

Note that, in this example, we have installed the servant locator for the empty category. This means that `locate` on our servant locator will be called for invocations to any of our Ice objects (because the empty category acts as the default). In effect, with this design, we are not using the `category` member of the object identity. This is fine, as long as all our servants all have the same, single interface.

4. The `Ice.Current` object is unique in this respect: for all other parameters, it is safe to store a reference to a parameter beyond the scope of the current invocations.

However, if we need to support several different interfaces in the same server, this simple strategy is no longer sufficient.

16.6.6 Using the category Member of the Object Identity

The simple example in the preceding section always instantiates a servant of type `PhoneEntryI`. In other words, the servant locator implicitly is aware of the type of servant the incoming request is for. This is not a very realistic assumption for most servers because, usually, a server provides access to objects with several different interfaces. This poses a problem for our `locate` implementation: somehow, we need to decide inside `locate` what type of servant to instantiate. You have several options for solving this problem:

- Use a separate object adapter for each interface type and use a separate servant locator for each object adapter.

This technique works fine, but has the down-side that each object adapter requires a separate transport endpoint, which is wasteful.

- Mangle a type identifier into the `name` component of the object identity.

This technique uses part of the object identity to denote what type of object to instantiate. For example, in our file system application, we have directory and file objects. By convention, we could prepend a ‘d’ to the identity of every directory and prepend an ‘f’ to the identity of every file. The servant locator then can use the first letter of the identity to decide what type of servant to instantiate:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current & c,
                         Ice::LocalObjectPtr & cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;
    std::string realId = c.id.name.substr(1);
    try {
        if(name[0] == 'd') {
            // The request is for a directory.
            //
            DirectoryDetails d = DB_lookup(realId);
            return new DirectoryI(d);
        } else {
            // The request is for a file.
        }
    }
}
```

```

        //
        FileDetails d = DB_lookup(realId);
        return new FileI(d);
    }
} catch (DatabaseNotFoundException &) {
    throw Ice::ObjectNotExistException(__FILE__, __LINE__);
}
}
}

```

While this works, it is awkward: not only do we need to parse the `name` member to work out what type of object to instantiate, but we also need to modify the implementation of `locate` whenever we add a new type to our application.

- Use the `category` member of the object identity to denote the type of servant to instantiate.

This is the recommended approach: for every interface type, we assign a separate identifier as the value of the `category` member of the object identity. (For example, we can use ‘d’ for directories and ‘f’ for files.) Instead of registering a single servant locator, we create two different servant locator implementations, one for directories and one for files, and then register each locator for the appropriate category:

```

class DirectoryLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr & cookie)
    {
        // Code to locate and instantiate a directory here...
    }

    virtual void finished(const Ice::Current & c,
                         const Ice::ObjectPtr & servant,
                         const Ice::LocalObjectPtr & cookie)
    {
    }

    virtual void deactivate(const std::string & category)
    {
    }
};

class FileLocator : public virtual Ice::ServantLocator {

```

```
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr & cookie)
    {
        // Code to locate and instantiate a file here...
    }

    virtual void finished(const Ice::Current & c,
                          const Ice::ObjectPtr & servant,
                          const Ice::LocalObjectPtr & cookie)
    {
    }

    virtual void deactivate(const std::string & category)
    {
    }
};

// ...

// Register two locators, one for directories and
// one for files.
//
adapter->addServantLocator(new DirectoryLocator(), "d");
adapter->addServantLocator(new FileLocator(), "f");
```

Yet another option is to use the `category` member of the object identity, but to use a single default servant locator (that is, a locator for the empty category). With this approach, all invocations go to the single default servant locator, and you can switch on the category value inside the implementation of the `locate` operation to determine which type of servant to instantiate. However, this approach is harder to maintain than the previous one; the `category` member of the Ice object identity exists specifically to support servant locators, so you might as well use it as intended.

16.6.7 Using Cookies

Occasionally, it can be useful to be able to pass information between `locate` and `finished`. For example, the implementation of `locate` could choose among a number of alternative database backends, depending on load or availability and, to properly finalize state, the implementation of `finish` might need to know which

database was used by `locate`. To support such scenarios, you can create a cookie in your `locate` implementation; the Ice run time passes the value of the cookie to `finished` after the operation invocation has completed. The cookie must be derived from `Ice::LocalObject` and can contain whatever state and member functions are useful to your implementation:

```
class MyCookie : public virtual Ice::LocalObject {
public:
    // Whatever is useful here...
};

typedef IceUtil::Handle<MyCookie> MyCookiePtr;

class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr & cookie)
    {
        // Code as before...

        // Allocate and initialize a cookie.
        //
        cookie = new MyCookie(...);

        return new PhoneEntryI();
    }

    virtual void finished(const Ice::Current & c,
                         const Ice::ObjectPtr & servant,
                         const Ice::LocalObjectPtr & cookie)
    {
        // Down-cast cookie to actual type.
        //
        MyCookiePtr mc = MyCookiePtr::dynamicCast(cookie);

        // Use information in cookie to clean up...
        //
        // ...
    }

    virtual void deactivate(const std::string & category);
};
```

16.7 Server Implementation Techniques

As we mentioned on page 384, instantiating a servant for each Ice object on server start-up is a viable design, provided that you can afford the amount of memory required by the servants, as well as the delay in start-up of the server. However, Ice supports more flexible mappings between Ice objects and servants; these alternate mappings allow you to precisely control the trade-off between memory consumption, scalability, and performance. We outline a few of the more common implementation techniques in this section.

16.7.1 Incremental Initialization

If you use a servant locator, the servant returned by `locate` is used only for the current request, that is, the Ice run time does not add the servant to the active servant map. Of course, this means that if another request comes in for the same Ice object, `locate` must again retrieve the object state and instantiate a servant. A common implementation technique is to add each servant to the ASM as part of `locate`. This means that only the first request for each Ice object triggers a call to `locate`; thereafter, the servant for the corresponding Ice object can be found in the ASM and the Ice run time can immediately dispatch another incoming request for the same Ice object without having to call the servant locator.

An implementation of `locate` to do this would look something like the following:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current & c,
                         Ice::LocalObjectPtr & cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try {
        d = DB_lookup(name);
    } catch (const DB_error &)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
}
```

```

    // We have the state, instantiate a servant.
    //
    Ice::ObjectPtr servant = new PhoneEntryI(d);

    // Add the servant to the ASM.
    //
    c.adapter->add(servant, c.id);           // NOTE: Incorrect!

    return servant;
}

```

This is almost identical to the implementation on page 391—the only difference is that we also add the servant to the ASM by calling `ObjectAdapter::add`. Unfortunately, this implementation is wrong because it suffers from a race condition. Consider the situation where we do not have a servant for a particular Ice object in the ASM, and two clients more or less simultaneously send a request for the same Ice object. It is entirely possible for the thread scheduler to schedule the two incoming requests such that the Ice run time completes the lookup in the ASM for both requests and, for each request, concludes that no servant is in memory. The net effect is that `locate` will be called twice for the same Ice object, and our servant locator will instantiate two servants instead of a single servant. Because the second call to `ObjectAdapter::add` will raise an `AlreadyRegisteredException`, only one of the two servants will be added to the ASM.

Of course, this is hardly the behavior we expect. To avoid the race condition, our implementation of `locate` must check whether a concurrent invocation has already instantiated a servant for the incoming request and, if so, return that servant instead of instantiating a new one. The Ice run time provides the `ObjectAdapter::find` operation to allow us to test whether an entry for a specific identity already exists in the ASM:

```

module Ice {
    local interface ObjectAdapter {
        // ...

        Object find(Identity id);

        // ...
    };
}

```

`find` returns the servant if it exists in the ASM and null, otherwise. Using this lookup function, together with a mutex, allows us to correctly implement `locate`.

The class definition of our servant locator now has a private mutex so we can establish a critical region inside locate:

```
class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr &);

    // Declaration of finished() and deactivate() here...

private:
    IceUtil::Mutex _m;
};
```

The locate member function locks the mutex and tests whether a servant is already in the ASM: if so, it returns that servant; otherwise, it instantiates a new servant and adds it to the ASM as before:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current & c,
                        Ice::LocalObjectPtr &)
{
    IceUtil::Mutex::Lock lock(_m);

    // Check if we have instantiated a servant already.
    //
    Ice::ObjectPtr servant = c.adapter.find(c.id);

    if(!servant) {      // We don't have a servant already

        // Instantiate a servant.
        //
        ServantDetails d;
        try {
            d = DB_lookup(c.id.name);
        } catch (const DB_error &) {
            throw Ice::ObjectNotExistException(__FILE__, __LINE__);
        }
        servant = new PhoneEntryI(d);

        // Add the servant to the ASM.
        //
        c.adapter->add(servant, c.id);
    }
}
```

```
        }

        return servant;
    }

The Java version of this locator is almost identical, but we use the
synchronized qualifier instead of a mutex to make locate a critical region:

synchronized public Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    // Check if we have instantiated a servant already.
    //
    Ice.Object servant = c.adapter.find(c.id);

    if(servant == null) { // We don't have a servant already

        // Instantiate a servant
        //
        ServantDetails d;
        try {
            d = DB.lookup(c.id.name);
        } catch (DB.error &)
        {
            throw new Ice.ObjectNotExistException();
        }
        servant = new PhoneEntryI(d);
    }

    return servant;
}
```

Using a servant locator that adds the servant to the ASM has a number of advantages:

- Servants are instantiated on demand, so the cost of initializing the servants is spread out over many invocations instead of being incurred all at once during server start-up.
- The memory requirements for the server are reduced because servants are instantiated only for those Ice objects that are actually accessed by clients. If clients only access a subset of the total number of Ice objects, the memory savings can be substantial.

In general, incremental initialization is beneficial if instantiating servants during start-up is too slow. The memory savings can be worthwhile as well but, as a rule, are realized only for comparatively short-lived servers: for long-running servers,

chances are that, sooner or later, every Ice object will be accessed by some client or another; in that case, there are no memory savings because we end up with an instantiated servant for every Ice object regardless.

16.7.2 Default Servants

A common problem with object-oriented middleware is scalability: servers frequently are used as front ends to large databases that are accessed remotely by clients. The servers' job is to present an object-oriented view to clients of a very large number of records in the database. Typically, the number of records is far too large to instantiate servants for even a fraction of the database records.

A common technique for solving this problem is to use *default servants*. A default servant is a servant that, for each request, takes on the persona of a different Ice object. In other words, the servant changes its behavior according to the object identity that is accessed by a request, on a per-request basis. In this way, it is possible to allow clients access to an unlimited number of Ice objects with only a single servant in memory.

Default servant implementations are attractive not only because of the memory savings they offer, but also because of the simplicity of implementation: in essence, a default servant is a facade [2] to the persistent state of an object in the database. This means that the programming required to implement a default servant is typically minimal: it simply consists of the code required to read and write the corresponding database records.

To create a default servant implementation, we need as many locators as there are non-abstract interfaces in the system. For example, for our file system application, we require two locators, one for directories and one for files. In addition, the object identities we create use the `category` member of the object identity to encode the type of interface of the corresponding Ice object. The value of the `category` field can be anything that identifies the interface, such as the '`d`' and '`f`' convention we used on page 396. Alternatively, you could use "`Directory`" and "`File`", or use the type ID of the corresponding interface, such as `::Filesystem::Directory` and `::Filesystem::File`. The `name` member of the object identity must be set to whatever identifier we can use to retrieve the persistent state of each directory and file from secondary storage. (For our filesystem application, we used a UUID as a unique identifier.)

The servant locators each return a singleton servant from their respective `locate` implementations. Here is the servant locator for directories:

```

class DirectoryLocator : public virtual Ice::ServantLocator {
public:
    DirectoryLocator() : _servant(new DirectoryI)
    {
    }

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr & cookie)
    {
        return _servant;
    }

    virtual void finished(const Ice::Current & c,
                          const Ice::ObjectPtr & servant,
                          const Ice::LocalObjectPtr & cookie);

    virtual void deactivate(const std::string & category);

private:
    Ice::ObjectPtr _servant;
};

```

Note that constructor of the locator instantiates a servant and returns that same servant from every call to `locate`. The implementation of the file locator is analogous to the one for directories. Registration of the servant locators proceeds as usual:

```

_adapter->addServantLocator(new DirectoryLocator, "d");
_adapter->addServantLocator(new FileLocator, "f");

```

All the action happens in the implementation of the operations, using the following steps for each operation:

1. Use the passed `Current` object to get the identity for the current request.
2. Use the `name` member of the identity to locate the persistent state of the servant on secondary storage. If no record can be found for the identity, throw an `ObjectNotExistException`.
3. Implement the operation to operate on that retrieved state (returning the state or updating the state as appropriate for the operation).

In pseudo-code, this might look something like the following:

```

Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current &c) const
{
    // Use the identity of the directory to retrieve

```

```
// its contents.  
DirectoryContents dc;  
try {  
    dc = DB_getDirectory(c.id.name);  
} catch(const DB_error &) {  
    throw Ice::ObjectNotExistException(__FILE__, __LINE__);  
}  
  
// Use the records retrieved from the database to  
// initialize return value.  
//  
FileSystem::NodeSeq ns;  
// ...  
  
return ns;  
}
```

Note that the servant implementation is completely stateless: the only state it operates on is the identity of the Ice object for the current request (and that identity is passed as part of the `Current` parameter). The price we have to pay for the unlimited scalability and reduced memory footprint is performance: default servants make a database access for every invoked operation which is obviously slower than caching state in memory, as part of a servant that has been added to the ASM. However, this does not mean that default servants carry an unacceptable performance penalty: database often provide sophisticated caching, so even though the operation implementations read and write the database, as long as they access cached state, performance may be entirely acceptable.

Overriding `ice_ping`

One issue you need to be aware of with default servants is the need to override `ice_ping`: the default implementation of `ice_ping` that the servant inherits from its skeleton class always succeeds. For servants that are registered with the ASM, this is exactly what we want; however, for default servants, `ice_ping` must fail if a client uses a proxy to a no-longer existent Ice object. To avoid getting successful `ice_ping` invocations for non-existent Ice objects, you must override `ice_ping` in the default servant. The implementation must check whether the object identity for the request denotes a still-existing Ice object and, if not, throw `ObjectNotExistException`:

```
void  
FilesystemDirectoryI::ice_ping(const Ice::Current &c)  
{  
    try {
```

```
    d = DB_lookup(c.id.name);
} catch (const DB_error &) {
    throw Ice::ObjectNotExistException(__FILE__, __LINE__);
}
}
```

It is good practice to override `ice_ping` if you are using default servants.

16.7.3 Hybrid Approaches and Caching

Depending on the nature of your application, you may be able to steer a middle path that provides better performance while keeping memory requirements low: if your application has a number of frequently-accessed objects that are performance-critical, you can add servants for those objects to the ASM. If you store the state of these objects in data members inside the servants, you effectively have a cache of these objects.

The remaining, less-frequently accessed objects can be implemented with a default servant. For example, in our file system implementation, we could choose to instantiate directory servants permanently, but to have file objects implemented with a default servant. This provides efficient navigation through the directory tree and incurs slower performance only for the (presumably less frequent) file accesses.

This technique could be augmented with a cache of recently-accessed files, along similar lines to the buffer pool used by the UNIX kernel [10]. The point is that you can combine use of the ASM with servant locators and default servants to precisely control the trade-offs among scalability, memory consumption, and performance to suit the needs of your application.

16.7.4 Servant Evictors

A variation on the previous theme and particularly interesting use of a servant locator is as an *evictor* [4]. An evictor is a servant locator that maintains a cache of servants:

- Whenever a request arrives (that is, `locate` is called by the Ice run time), the evictor checks to see whether it can find a servant for the request in its cache. If so, it returns the servant that is already instantiated in the cache; otherwise, it instantiates a servant and adds it to the cache.
- The cache is a queue that is maintained in least-recently used (LRU) order: the least-recently used servant is at the tail of the queue, and the most-recently

used servant is at the head of the queue. Whenever a servant is returned from or added to the cache, it is moved from its current queue position to the head of the queue, that is, the “newest” servant is always at the head, and the “oldest” servant is always at the tail.

- The queue has a configurable length that corresponds to how many servants will be held in the cache; if a request arrives for an Ice object that does not have a servant in memory and the cache is full, the evictor removes the least-recently used servant at the tail of the queue from the cache in order to make room for the servant about to be instantiated at the head of the queue.

Figure 16.2 illustrates an evictor with a cache size of five after five invocations have been made, for object identities 1 to 5, in that order.

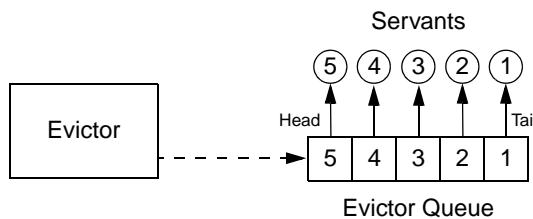


Figure 16.2. An evictor after five invocations for object identities 1 to 5.

At this point, the evictor has instantiated five servants, and has placed each servant onto the evictor queue. Because requests were sent by the client for object identities 1 to 5 (in that order), servant 5 ends up at the head of the queue (at the most-recently used position), and servant 1 ends up at the tail of the queue (at the least-recently used position).

Assume that the client now sends a request for servant 3. In this case, the servant is found on the evictor queue and moved to the head position. The resulting ordering is shown in Figure 16.3.

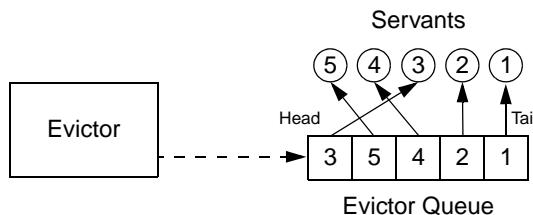


Figure 16.3. The evictor from Figure 16.2 after accessing servant 3.

Assume that the next client request is for object identity 6. The evictor queue is fully populated, so the evictor creates a servant for object identity 6, places that servant at the head of the queue, and evicts the servant with identity 1 (the least-recently used servant) at the tail of the queue, as shown in Figure 16.4.

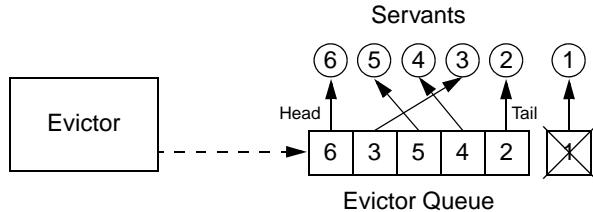


Figure 16.4. The evictor from Figure 16.3 after evicting servant 1.

The evictor pattern combines the advantages of the ASM with the advantages of a default servant: provided that the cache size is sufficient to hold the working set of servants in memory, most requests are served by an already instantiated servant, without incurring the overhead of creating a servant and accessing the database to initialize servant state. By setting the cache size, you can control the trade-off between performance and memory consumption as appropriate for your application.

The following sections show how to implement an evictor in both C++ and Java. (You can also find the source code for the evictor with the code examples for this book in the Ice distribution.)

Creating an Evictor Implementation in C++

The evictor we show here is designed as an abstract base class: in order to use it, you derive an object from the `EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definitions as follows:

```
class EvictorBase : public Ice::ServantLocator {
public:

    EvictorBase(int size = 1000);

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr & cookie);
    virtual void finished(const Ice::Current & c,
                         const Ice::ObjectPtr & servant,
                         const Ice::LocalObjectPtr & cookie);
```

```
virtual void deactivate(const std::string & category);

virtual Ice::ObjectPtr add(const Ice::Current & c,
                           Ice::LocalObjectPtr & cookie) = 0;
virtual void evict(const Ice::ObjectPtr & servant,
                   const Ice::LocalObjectPtr & cookie) = 0;

private:
    // ...
};

typedef IceUtil::Handle<EvictorBase> EvictorBasePtr;
```

Note that the evictor has a constructor that sets the size of the queue, with a default argument to set the size to 1000.

The `locate`, `finished`, and `deactivate` functions are inherited from the `ServantLocator` base class; these functions implement the logic to maintain the queue in LRU order and to add and evict servants as needed.

The `add` and `evict` functions are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are pure virtual, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant. This allows `evict` to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map does not only store servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue. Storing the queue position is not strictly necessary—we store the position for efficiency reasons because it allows us to locate a servant’s position in the queue in constant time instead of having to search through the queue in order to maintain its LRU property.

3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The need for the use count deserves some extra explanation: suppose a client invokes a long-running operation on an Ice object with identity I . In response, the evictor adds a servant for I to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity I gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object I is still executing, the servant for I could be evicted while it is still executing the original request.

By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object I , the evictor would have no idea that a servant for I is still around and would add a second servant for I . However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

The evictor queue does not store the identity of the servant. Instead, the entries on the queue are iterators into the evictor map. This is useful when the time comes to evict a servant: instead of having to search the map for the identity of the servant to be evicted, we can simply delete the map entry that is pointed at by the iterator at the tail of the queue. We can get away with storing an iterator into the evictor queue as part of the map, and storing an iterator into the evictor map as part of the queue because both `std::list` and `std::map` do not invalidate iterators when we add or delete entries (except for invalidating iterators that point at a deleted entry, of course).

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

The leads to the following definitions in the private section of our evictor:

```
class EvictorBase : public Ice::ServantLocator {
public:
    // ...

private:
    struct EvictorEntry;
    typedef IceUtil::Handle<EvictorEntry> EvictorEntryPtr;

    typedef std::map<Ice::Identity, EvictorEntryPtr> EvictorMap;
    typedef std::list<EvictorMap::iterator> EvictorQueue;

    struct EvictorEntry : public Ice::LocalObject {
        Ice::ObjectPtr servant;
        Ice::LocalObjectPtr userCookie;
        EvictorQueue::iterator pos;
        int useCount;
    };

    struct EvictorCookie : public Ice::LocalObject {
        EvictorEntryPtr entry;
    };
    typedef IceUtil::Handle<EvictorCookie> EvictorCookiePtr;

    EvictorMap _map;
    EvictorQueue _queue;
    Ice::Int _size;

    IceUtil::Mutex _mutex;

    void evictServants();
};
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. In addition, we use a private `_mutex` data member so we can correctly serialize access to the evictor's data structures.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit—we will discuss this function in more detail shortly.

The implementation of the constructor is trivial. The only point of note is that we ignore negative sizes:⁵

```
EvictorBase::EvictorBase(Ice::Int size) : _size(size)
{
    if (_size < 0)
        _size = 1000;
}
```

Almost all the action of the evictor takes place in the implementation of `locate`:

```
Ice::ObjectPtr
EvictorBase::locate(const Ice::Current &c,
                    Ice::LocalObjectPtr &cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    // Create a cookie.
    //
    EvictorCookiePtr ec = new EvictorCookie;
    cookie = ec;

    // Check if we have a servant in the map already.
    //
    EvictorMap::iterator i = _map.find(c.id);
    bool newEntry = i == _map.end();
    if(!newEntry) {
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        ec->entry = i->second;
        _queue.erase(ec->entry->pos);
    } else {
        // We do not have an entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        ec->entry = new EvictorEntry;
        ec->entry->servant
            = add(c, ec->entry->userCookie); // Down-call
        if(!ec->entry->servant)
        {
            throw Ice::ObjectNotExistException(__FILE__, __LINE__);
        }
        ec->entry->useCount = 0;
    }
}
```

-
5. We could have stored the size as a `size_t` instead. However, for consistency with the Java implementation, which cannot use unsigned integers, we use `Ice::Int` to store the size.

```
i = _map.insert(std::make_pair(c.id, ec->entry)).first;
}

// Increment the use count of the servant and enqueue
// the entry at the front, so we get LRU order.
//
++(ec->entry->useCount);
ec->entry->pos = _queue.insert(_queue.begin(), i);

return ec->entry->servant;
}
```

The first step in `locate` is to lock the `_mutex` data member. This protects the evictor's data structures from concurrent access. The next step is to create the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`. The cookie contains a smart pointer to an evictor entry, of type `EvictorEntryPtr`. This is also the value type of our map entries, so we do not store two copies of the same information redundantly—instead, smart pointers ensure that a single copy is shared by both the cookie and the map.

The next step is to look in the evictor map to see whether we already have an entry for this object identity. If so, we initialize the cookie's smart pointer with that entry and remove the entry from its current queue position.

Otherwise, we do not have an entry for this object identity yet, so we have to create one. The code creates a new evictor entry, and then calls `add` to get a new servant. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we throw an `ObjectNotExistException` to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The last few lines of `locate` add the entry for the current request to the head of the evictor queue to maintain its LRU property, increment the use count of the entry, and finally return the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

```

void
EvictorBase::finished(const Ice::Current &,
                      const Ice::ObjectPtr &,
                      const Ice::LocalObjectPtr &cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    EvictorCookiePtr ec = EvictorCookiePtr::dynamicCast(cookie);

    // Decrement use count and check if
    // there is something to evict.
    //
    --(ec->entry->useCount);
    evictServants();
}

```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

```

void
EvictorBase::evictServants()
{
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    for (int i = static_cast<int>(_map.size() - _size);
         i > 0; --i) {
        EvictorQueue::reverse_iterator p = _queue.rbegin();
        if((*p)->second->useCount == 0) {
            evict((*p)->second->servant, (*p)->second->userCookie);
            EvictorMap::iterator pos = *p;
            _queue.erase((*p)->second->pos);
            _map.erase(pos);
        }
    }
}

```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run

time guarantees to call deactivate only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

```
void
EvictorBase::deactivate(const std::string & category)
{
    IceUtil::Mutex::Lock lock(_mutex);

    _size = 0;
    evictServants();
}
```

Creating an Evictor Implementation in Java

The evictor we show here is designed as an abstract base class: in order to use it, you derive an object from the `Evictor.EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definitions as follows:

```
package Evictor;

public abstract class EvictorBase
    extends Ice.LocalObjectImpl implements Ice.ServantLocator
{
    public
    EvictorBase()
    {
        _size = 1000;
    }

    public
    EvictorBase(int size)
    {
        _size = size < 0 ? 1000 : size;
    }

    public abstract Ice.Object
    add(Ice.Current c, Ice.LocalObjectHolder cookie);

    public abstract void
    evict(Ice.Object servant, Ice.LocalObject cookie);

    synchronized public final Ice.Object
    locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
```

```
// ...
}

synchronized public final void
finished(Ice.Current c, Ice.Object o, Ice.LocalObject cookie)
{
    // ...
}

synchronized public final void
deactivate(String category)
{
    // ...
}

// ...

private int _size;
}
```

Note that the evictor has constructors to set the size of the queue, with a default size of 1000.

The `locate`, `finished`, and `deactivate` methods are inherited from the `ServantLocator` base class; these methods implement the logic to maintain the queue in LRU order and to add and evict servants as needed. The methods are synchronized, so the evictor's internal data structures are protected from concurrent access.

The `add` and `evict` methods are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are abstract, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant. This allows `evict` to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map does not only store servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue.
3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The last two points deserve some extra explanation.

- The evictor queue must be maintained in least-recently used order, that is, every time an invocation arrives and we find an entry for the identity in the evictor map, we also must locate the servant's identity on the evictor queue and move it to the front of the queue. However, scanning for that entry is inefficient because it requires $O(n)$ time. To get around this, we store an iterator in the evictor map that marks the corresponding entry's position in the evictor queue. This allows us to dequeue the entry from its current position and enqueue it at the head of the queue in $O(1)$ time.

Unfortunately, the various lists provided by `java.util` do not allow us to keep an iterator to a list position without invalidating that iterator as the list is updated. To deal with this, we use a special-purpose linked list implementation, `Evictor.LinkedList`, that does not have this limitation.

`LinkedList` has an interface similar to `java.util.LinkedList` but does not invalidate iterators other than iterators that point at an element that is removed. For brevity, we do not show the implementation of this list here—you can find the implementation in the code examples for this book in the Ice distribution.

- We maintain a use count as part of the map in order to avoid incorrect eviction of servants. Suppose a client invokes a long-running operation on an Ice object with identity I . In response, the evictor adds a servant for I to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity I gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object I is still

executing, the servant for *I* could be evicted while it is still executing the original request.

By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*.

However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

```
package Evictor;

public abstract class EvictorBase
    extends Ice.LocalObjectImpl implements Ice.ServantLocator
{
    // ...

    private class EvictorEntry
    {
        Ice.Object servant;
        Ice.LocalObject userCookie;
        java.util.Iterator pos;
        int useCount;
    }

    private class EvictorCookie extends Ice.LocalObjectImpl
    {
        public EvictorEntry entry;
    }

    private void evictServants()
    {
        // ...
    }
}
```

```
    }

    private java.util.Map _map = new java.util.HashMap();
    private Evictor.LinkedList _queue = new Evictor.LinkedList();
    private int _size;
}
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. The map key is the identity of the Ice object, and the lookup value is of type `EvictorEntry`. The queue simply stores identities, of type `Ice::Identity`.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit—we will discuss this function in more detail shortly.

Almost all the action of the evictor takes place in the implementation of `locate`:

```
synchronized public final Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    // Make a copy of the ID. We need to do this because
    // Ice.Current.id is the same reference on every call: its
    // contents change, but it is the same object every time. We
    // need a copy to insert into the queue and the map.
    //
    Ice.Identity idCopy = new Ice.Identity();
    idCopy.name = c.id.name;
    idCopy.category = c.id.category;

    // Create a cookie.
    //
    EvictorCookie ec = new EvictorCookie();
    cookie.value = ec;

    // Check if we a servant in the map already.
    //
    ec.entry = (EvictorEntry)_map.get(idCopy);
    boolean newEntry = ec.entry == null;
    if(!newEntry) {
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        ec.entry.pos.remove();
    } else {
        // We do not have entry. Ask the derived class to
    }
}
```

```

// instantiate a servant and add a new entry to the map.
//
ec.entry = new EsvictorEntry();
Ice.LocalObjectHolder cookieHolder
    = new Ice.LocalObjectHolder();
ec.entry.servant = add(c, cookieHolder); // Down-call
if(ec.entry.servant == null) {
    throw new Ice.ObjectNotExistException();
}
ec.entry.userCookie = cookieHolder.value;
ec.entry.useCount = 0;
_map.put(idCopy, ec.entry);
}

// Increment the use count of the servant and enqueue
// the entry at the front, so we get LRU order.
//
++(ec.entry.useCount);
_queue.addFirst(idCopy);
ec.entry.pos = _queue.iterator();
ec.entry.pos.next(); // Position the iterator on the element.

return ec.entry.servant;
}

```

The first step is to create a copy of the object identity that is passed in `Ice.Current`. (As mentioned on page 394, `Ice.Current` and members of `Ice.Current` are assigned to prior to passing them to application code instead of allocating a new object for each call for efficiency reasons.)

Next, the code creates a cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`. The cookie contains an evictor entry, of type `EsvictorEntry`.

We now look for an existing entry in the evictor map, using the object identity as the key. If we have an entry in the map already, we dequeue the corresponding identity from the evictor queue. (The `pos` member of `EsvictorEntry` is an iterator that marks that entry's position in the evictor queue.)

Otherwise, we do not have an entry in the map, so we create a new one and call the `add` method. This is a down-call to the concrete class that will be derived from `EsvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we throw an `ObjectNotExistException` to let the Ice run time know that no servant could be found for the current request. If

add succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The final few lines of code increment the entry's use count, add the entry at the head of the evictor queue, and store the entry's position in the queue before returning the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

```
synchronized public final void
finished(Ice.Current c, Ice.Object o, Ice.LocalObject cookie)
{
    EvictorCookie ec = (EvictorCookie)cookie;

    // Decrement use count and check if
    // there is something to evict.
    //
    --(ec.entry.useCount);
    evictServants();
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

```
private void evictServants()
{
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    for(int i = _map.size() - _size; i > 0; --i) {
        java.util.Iterator p = _queue.iterator();
        Ice.Identity id = (Ice.Identity)p.next();
        EvictorEntry e = (EvictorEntry)_map.get(id);
        if(e.useCount == 0) {
            evict(e.servant, e.userCookie); // Down-call
            p.remove();
            _map.remove(id);
        }
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function

in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

```
synchronized public final void
deactivate(String category)
{
    _size = 0;
    evictServants();
}
```

Using Servant Evictors

Using a servant evictor is simply a matter of deriving a class from `EvictorBase` and implementing the `add` and `evict` methods. You can turn a servant locator into an evictor by simply taking the code that you wrote for `locate` and placing it into `add`—`EvictorBase` then takes care of maintaining the cache in least-recently used order and evicting servants as necessary. Unless you have clean-up requirements for your servants (such as closing network connections or database handles), the implementation of `evict` can be left empty.

One of the nice aspects of evictors is that you do not need to change anything in your servant implementation: the servants are ignorant of the fact that an evictor is in use. This makes it very easy to add an evictor to an already existing code base with little disturbance of the source code.

Evictors can provide substantial performance improvements over default servants: especially if initialization of servants is expensive (for example, because servant state must be initialized by reading from a network), an evictor performs much better than a default servant, while keeping memory requirements low.

As we will see in XREF, evictors can also be useful to get rid of servants that have been abandoned by clients.

16.8 The `Ice::Context` Parameter

In Sections 6.11.1 and 8.11.1, methods on a proxy are overloaded with a trailing parameter of type `const Ice::Context &` (C++) or `java.util.Map` (Java). The Slice definition of this parameter is as follows:

```
module Ice {
    local dictionary<string, string> Context;
};
```

As you can see, a context is a dictionary that maps strings to strings or, conceptually, a context is a collection of name–value pairs. The contents of this dictionary (if any) are implicitly marshaled with every request to the server, that is, if the client populates a context with a number of name–value pairs and uses that context for an invocation, the name–value pairs that are sent by the client are available to the server.

On the server side, the operation implementation can access the received Context via the ctx member of the Ice::Current parameter (see Section 16.5) and extract the name–value pairs that were sent by the client.

16.8.1 Passing Context Explicitly

Contexts provide a means of sending an unlimited number of parameters from client to server without having to mention those parameters in the signature of an operation. For example, consider the following definition:

```
struct Address {
    // ...
};

interface Person {
    string setAddress(Address a);
    // ...
};
```

Assuming that the client has a proxy to a Person object, it could do something along the following lines:

```
PersonPrx p = ...;
Address a = ...;

Ice::Context ctx;
ctx["write policy"] = "immediate";

p->setAddress(a, ctx);
```

In Java, the same code would look as follows:

```
Person p = ...;
Address a = ...;

java.util.Map ctx = new java.util.HashMap();
ctx.put("write policy", "immediate");

p.setAddress(a, ctx);
```

On the server side, we can extract the policy value set by the client to influence how the implementation of `setAddress` works. A C++ implementation might look like:

```
void
PersonI::setAddress(const Address &a, const Ice::Current &c)
{
    Ice::Context::const_iterator i = c.ctx.find("write policy");
    if (i != c.ctx.end() && i->second == "immediate") {

        // Update the address details and write through to the
        // data base immediately...

    } else {

        // Write policy was not set (or had a bad value), use
        // some other database write strategy.
    }
}
```

For this example, the server examines the value of the context with the key "write policy" and, if that value is "immediate", writes the update sent by the client straight away; if the write policy is not set or contains a value that is not recognized, the server presumably applies a more lenient write policy (such as caching the update in memory and writing it later). The Java version of the operation implementation is essentially identical, so we do not show it here.

16.8.2 Passing Context Implicitly

Instead of passing a context explicitly with an invocation, you can also use an *implicit* context. Implicit contexts allow you to set a context on a particular proxy once and, thereafter, whenever you use that proxy to invoke an operation, the previously-set context is sent with each invocation. The proxy base class provides a member function, `ice_newContext`, to do this:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
            public:
                Ice::ObjectPrx
                    ice_newContext(const Ice::Context &) const;
                // ...
            };
        };
    }
}
```

For Java, the corresponding function is:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_newContext(java.util.Map newContext);
    // ...
}
```

`ice_newContext` creates a new proxy that implicitly stores the passed context. Note that the return type of `ice_newContext` is `ObjectPrx`, that is, before you can use the newly-created proxy, you must down-cast it to the correct type. For example, in C++:

```
Ice::Context ctx;
ctx["write policy"] = "immediate";

PersonPrx p1 = ...;
PersonPrx p2 = PersonPrx::uncheckedCast(p1->ice_newContext(ctx));

Address a = ...;

p1->setAddress(a);           // Sends no context

p2->setAddress(a);           // Sends ctx implicitly

Ice::Context ctx2;
ctx2["write policy"] = "delayed";

p2->setAddress(a, ctx2); // Sends ctx2
```

As the example illustrates, once we have created the `p2` proxy, any invocation via `p2` implicitly sends the previously-set context. The final line of the example illustrates that it is possible to explicitly send a context for an invocation even if the

proxy has an implicit context—an explicit context always overrides any implicit context.

16.8.3 Retrieving Implicit Context

You can retrieve the implicit context of a proxy by calling `ice_getContext`. The call returns the currently-set context of the proxy. (If a proxy has no implicit context, the returned dictionary is empty.) For C++, the signature is:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
            public:
                Ice::Context ice_getContext() const;
                // ...
        };
    }
}
```

For Java, the signature is:

```
package Ice;

public interface ObjectPrx {
    java.util.Map ice_getContext();
    // ...
}
```

16.8.4 Context Use Cases

The purpose of `Ice::Context` is to permit services to be added to Ice that require some contextual information with every request. Such contextual information can be used by services such as a transaction service (to provide the context of a currently established transaction) or a security service (to provide an authorization token to the server). IceStorm (see Chapter 27) uses the context to provide an optional `cost` parameter to the service that influences how the service propagates messages to down-stream subscribers.

In general, services that require such contextual information can be implemented much more elegantly using contexts because this hides explicit Slice parameters that would otherwise have to be supplied by the application programmer with every call.

In addition, contexts, because they are optional, permit a single Slice definition to apply to implementations that use the context as well as to implementations

that do not use it. In this way, to add transactional semantics to an existing service, you do not need to modify the Slice definitions to add extra parameters to all operations. This not only is convenient for clients but also prevents the type system from being split into two halves: without contexts, we would need different Slice definitions for transactional and non-transactional implementations of what, conceptually, is a single service.

Finally, implicit contexts permit context information to be passed by through intermediate parts of a system without cooperation of those intermediate parts. For example, suppose you set an implicit context on a proxy and then pass that proxy to another system component. When that component uses the proxy to invoke an operation, the implicit context will still be sent. In other words, implicit contexts allow you to transparently propagate information via intermediaries that are ignorant of the presence of any context.

16.8.5 A Word of Warning

Contexts are a powerful mechanism for transparent propagation of context information, *if used correctly*. In particular, you may be tempted to use contexts as a means of versioning an application as it evolves over time. For example, version 2 of your application may accept two parameters on an operation that, in version 1, used to accept only a single parameter. Using contexts, you could supply the second parameter as a name–value pair to the server and avoid changing the Slice definition of the operation in order to maintain backward compatibility.

We *strongly* urge you to resist any temptation to use contexts in this manner. The strategy is fraught with problems:

- Missing context

There is nothing that would compel a client to actually send a context when the server expects to receive a context: if a client forgets to send a context, the server, somehow, has to make do without it (or throw an exception).

- Missing or incorrect keys

Even if the client does send a context, there is no guarantee that it has set the correct key. (For example, a simple spelling error can cause the client to send a value with the wrong key.)

- Incorrect values

The value of a context is a string, but the application data that is to be sent might be a number, or it might be something more complex, such as a structure with several members. This forces you to encode the value into a string

and decode the value again on the server side. Such parsing is tedious and error prone, and far less efficient than sending strongly-typed parameters. In addition, the server has to deal with string values that fail to decode correctly (for example, because of an encoding error made by the client).

None of the preceding problems can arise if you use proper Slice parameters: parameters cannot be accidentally omitted and they are strongly typed, making it much less likely for the client to accidentally send a meaningless value.

If you are concerned about how to evolve an application over time without breaking backward compatibility, Ice facets are better suited to this task (see Chapter 18). Contexts are meant to be used to transmit simple tokens (such as a transaction identifier) for services that cannot be reasonably implemented without them; you should restrict your use of contexts to that purpose and resist any temptation to use contexts for any other purpose.

Finally, be aware that, if a request is routed via one or more Ice routers, contexts may be dropped by intermediate routers if they consider them illegal. This means that, in general, you cannot rely on an arbitrary context value that is created by an application to actually still be present when a request arrives at the server—only those context values that are known to routers and that are considered legitimate are passed on. It follows that you should not abuse contexts to pass things that really should be passed as parameters.

16.9 Invocation Timeouts

Remote invocations made by clients are synchronous and blocking: the invocation does not complete on the client side until the server has finished processing it. Occasionally, it is useful to be able to force an invocation to terminate after some time, even if it has not completed. Proxies provide the `ice_timeout` operation for this:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
            public:
                Ice::ObjectPrx ice_timeout(Ice::Int t) const;
                // ...
        };
    }
}
```

For Java, the corresponding method is:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_timeout(int t);
    // ...
}
```

The `ice_timeout` operation creates a proxy with a timeout from an existing proxy. For example:

```
Filesystem::FilePrx myFile = ....;
FileSystem::FilePrx timeoutFile
    = FileSystem::FilePrx::uncheckedCast(
        myFile->ice_timeout(5000));

try {
    Lines text = timeoutFile->read();      // Read with timeout
} catch(const Ice::TimeoutException &) {
    cerr << "invocation timed out" << endl;
}

Lines text = myFile->read();                  // Read without timeout
```

The parameter to `ice_timeout` determines the timeout value in milliseconds. A value of `-1` indicates no timeout. In the preceding example, the timeout is set to five seconds; if an invocation of `read` via the `timeoutFile` proxy does not complete within five seconds, the operation terminates with an `Ice::TimeoutException`. On the other hand, invocations via the `myFile` proxy are unaffected by the timeout, that is, `ice_timeout` sets the timeout on a per-proxy basis.

The timeout value set on a proxy affects all networking operations: reading and writing of data as well as opening and closing of connections. If any of these operations does not complete within the timeout, the client receives an exception. Timeouts that expire during reading or writing of data are indicated by a `TimeoutException`. For opening and closing of connections, the Ice run time reserves separate exceptions:

- `ConnectTimeoutException`

This exception indicates that a connection could not be established within the specified time.

- `CloseTimeoutException`

This exception indicates that a connection could not be closed within the specified time.

The Ice run time provides the `Ice.Override.Timeout` property (see Appendix C). This property allows you to replace the default timeout value of `-1` (no timeout) with a different value. Setting this property affects all proxies for which you have not otherwise set a timeout with `ice_timeout`.

Note that timeouts are “soft” timeouts, in the sense that they are not precise, real-time timeouts. (The precision is limited by the capabilities of the underlying operating system.) You should also be aware that timeouts are considered fatal error conditions by the Ice run time: for example, timeouts result in connection closure on the client side. Timeouts are meant to be used to prevent a client from blocking indefinitely in case something has gone wrong with the server; they are not meant as a mechanism to routinely abort requests that take longer than intended.

16.10 Oneway Invocations

As mentioned in Chapter 2, the Ice run time supports *oneway* invocations. A oneway invocation is sent on the client side by writing the request to the client’s local transport buffers; the invocation completes and returns control to the application code as soon as it has been accepted by the local transport. Of course, this means that a oneway invocation is unreliable: it may never be sent (for example, because of a network failure) or it may not be accepted in the server (for example, because the target object does not exist). If anything goes wrong, the client-side application code does not receive any notification of the failure; the only errors that are reported to the client are local errors that occur on the client side during call invocation (such as failure to establish a connection, for example).

Oneway invocations are received and processed on the server side like any other incoming request. In particular, there is no way for the server-side application code to distinguish a oneway invocation from a twoway invocation, that is, oneway invocation is transparent on the server side.

Oneway invocations do not incur any return traffic from the server to the client: the server never sends a reply message in response to a oneway invocation (see Chapter 19). This means that oneway invocations can result in large efficiency gains, especially for large numbers of small messages, because the client

does not have to wait for the reply to each message to arrive before it can send the next message.

In order to be able to invoke an operation as oneway, two conditions must be met:

- The operation must have a `void` return type, must not have any `out`-parameters, and must not have an exception specification.

This requirement reflects the fact that the server does not send a reply for a oneway invocation to the client: without such a reply, there is no way to return any values or exceptions to the client.

If you attempt to invoke an operation that returns values to the client as a oneway operation, the Ice run time throws a `TwoWayOnlyException`.

- The proxy on which the operation is invoked must support a stream-oriented transport (such as TCP or SSL).

Oneway invocations require a stream-oriented transport. (To get something like a oneway invocation for datagram transports, you need to use a datagram invocation—see Section 16.11.)

If you attempt to create a oneway proxy for an object that does not offer a stream-oriented transport, the Ice run time throws a `NoEndpointException`.

Despite their theoretical unreliability, in practice, oneway invocations are reliable (but not infallible): they are sent via a stream-oriented transport, so they cannot get lost unless the connection fails entirely. In particular, the transport uses its usual flow control, so the client cannot overrun the server with messages. On the client-side, the Ice run time will block if the client's transport buffers fill up, so the client-side application code cannot overrun its local transport.

Consequently, oneway invocations normally do not block the client-side application code and return immediately, provided that the client does not consistently generate messages faster than the server can process them. If the rate at which the client invokes operations exceeds the rate at which the server can process them, the client-side application code will eventually block in an operation invocation until sufficient room is available in the client's transport buffers to accept the invocation.

Regardless of whether the client exceeds the rate at which the server can process incoming oneway invocations, the execution of oneway invocations in the server proceeds asynchronously: the client's invocation completes before the message even arrives at the server.

One thing you need to keep in mind about oneway invocations is that they may appear to be reordered in the server: because oneway invocations are sent via a

stream-oriented transport, they are guaranteed to be received in the order in which they were sent. However, the server dispatches each invocation in its own thread; because threads are scheduled preemptively, this may cause an invocation sent later by the client to be dispatched and executed before an invocation that was sent earlier.

For this reason, oneway invocations are usually best suited to simple updates that are otherwise stateless (that is, do not depend on the surrounding context or the state established by previous invocations).

Creating Oneway Proxies

Ice selects between twoway, oneway, and datagram (see Section 16.11) invocations via the proxy that is used to invoke the operation. By default, all proxies are created as twoway proxies. To invoke an operation as oneway, you must create a separate proxy for oneway dispatch from a twoway proxy.

For C++, all proxies are derived from a common `IceProxy::Ice::Object` class (see Section 6.11.1). The proxy base class contains a method to create a oneway proxy, called `ice_oneway`:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
            public:
                Ice::ObjectPrx ice_oneway() const;
                // ...
        };
    }
}
```

For Java, proxies are derived from the `Ice.ObjectPrx` interface (see Section 8.11.2) and the definition of `ice_oneway` is:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_oneway();
    // ...
}
```

We can call `ice_oneway` to create a oneway proxy and then use the proxy to invoke an operation as follows. (We show the C++ version here—the Java version is analogous.)

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a oneway proxy.
//
Ice::ObjectPrx oneway;
try {
    oneway = o->ice_oneway();
} catch (const Ice::NoEndPointException &) {
    cerr << "No endpoint for oneway invocations" << endl;
}

// Down-cast to actual type.
//
PersonPrx onewayPerson = PersonPrx::uncheckedCast(oneyway);

// Invoke an operation as oneway.
//
try {
    onewayPerson->someOp();
} catch (const Ice::TwowayOnlyException &) {
    cerr << "someOp() is not oneway" << endl;
}
```

Note that we use an `uncheckedCast` to down-cast the proxy from `ObjectPrx` to `PersonPrx`: for a oneway proxy, we cannot use a `checkedCast` because a `checkedCast` requires a reply from the server but, of course, a oneway proxy does not permit that reply. If instead you want to use a safe down-cast, you can first down-cast the twoway proxy to the actual object type and then obtain the oneway proxy:

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Safe down-cast to actual type.
//
PersonPrx person = PersonPrx::checkedCast(o);

if (person) {
    // Get a oneway proxy.
    //
    PersonPrx onewayPerson;
    try {
        onewayPerson
            = PersonPrx::uncheckedCast(person->ice_oneway());
    } catch (const Ice::NoEndPointException &) {
        cerr << "No endpoint for oneway invocations" << endl;
}
```

```
}

// Invoke an operation as oneway.
//
try {
    onewayPerson->someOp();
} catch (const Ice::TwowayOnlyException &) {
    cerr << "someOp() is not oneway" << endl;
}
}
```

Note that, while the second version of this code is somewhat safer (because it uses a safe down-cast), it is also slower (because the safe down-cast incurs the cost of an additional twoway message).

16.11 Datagram Invocations

Datagram invocations are the equivalent of oneway invocations for datagram transports. As for oneway invocations, datagram invocations can be sent only for operations that have a `void` return type and do not have `out`-parameters or an exception specification. (Attempts to use a datagram invocation with an operation that does not meet these criteria result in a `TwowayOnlyException`.) In addition, datagram invocations can only be used if the proxy's endpoints include at least one UDP transport; otherwise, the Ice run time throws a `NoEndpointException`.

The semantics of datagram invocations are similar to oneway invocations: no return traffic flows from the server to the client and proceed asynchronously with respect to the client; a datagram invocation completes as soon as the client's transport has accepted the invocation into its buffers. However, datagram invocations have additional error semantics:

- Individual invocations may be lost or received out of order.

On the wire, datagram invocations are sent as true datagrams, that is, individual datagrams may be lost, or arrive at the server out of order. As a result, not only may operations be dispatched out of order, an individual invocation out of a series of invocations may be lost. (This cannot happen for oneway invocations because, if a connection fails, *all* invocations are lost once the connection breaks down.)

- UDP packets may be duplicated by the transport.

Because of the nature of UDP routing, it is possible for datagrams to arrive in duplicate at the server. This means that, for datagram invocations, Ice does *not* guarantee at-most-once semantics (see page 14): if UDP datagrams are duplicated, the same invocation may be dispatched more than once in the server.

- UDP packets are limited in size.

The maximum size of an IP datagram is 65,535 bytes. Of that, the IP header consumes 20 bytes, and the UDP header consumes 8 bytes, leaving 65,507 bytes as the maximum payload. If the marshaled form of an invocation, including the Ice request header (see Chapter 19) exceeds that size, the invocation is lost. (Exceeding the size limit for a UDP datagram is indicated to the application by a `DatagramLimitException`.)

Because of their unreliable nature, datagram invocations are best suited to simply update messages that are otherwise stateless. In addition, due to the high probability of loss of datagram invocations over wide area networks, you should restrict use of datagram invocations to local area networks, where they are less likely to be lost. (Of course, regardless of the probability of loss, you must design your application such that it can tolerate lost or duplicated messages.)

Creating Datagram Proxies

To create a datagram proxy, you must call `ice_datagram` on the proxy, for example:

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a datagram proxy.
//
Ice::ObjectPrx datagram;
try {
    datagram = o->ice_datagram();
} catch (const Ice::NoEndPointException &) {
    cerr << "No endpoint for datagram invocations" << endl;
}

// Down-cast to actual type.
//
PersonPrx datagramPerson = PersonPrx::uncheckedCast(datagram);

// Invoke an operation as a datagram.
//
try {
```

```

        datagramPerson->someOp();
    } catch (const Ice::TwowayOnlyException &) {
        cerr << "someOp() is not oneway" << endl;
    }
}

```

As for the oneway example in Section 16.10, you can choose to first do a safe down-cast to the actual type of interface and then obtain the datagram proxy, rather than relying on an unsafe down-cast, as shown here. However, doing so may be disadvantageous for two reasons:

- Safe down-casts are sent via a stream-oriented transport. This means that using a safe down-cast will result in opening a connection for the sole purpose of verifying that the target object has the correct type. This is expensive if all the other traffic to the object is sent via datagrams.
- If the proxy does not offer a stream-oriented transport, the `checkedCast` fails with a `NoEndpointException`, so you can use this approach only for proxies that offer both a UDP endpoint and a TCP/IP and/or SSL endpoint.

16.12 Batched Invocations

Oneway and datagram invocations are normally sent as a separate message, that is, the Ice run time sends the oneway or datagram invocation to the server immediately, as soon as the client makes the call. If a client sends a number of oneway or datagram invocations in succession, the client-side run time traps into the OS kernel for each message, which is expensive. In addition, each message is sent with its own message header (see Chapter 19), that is, for n messages, the bandwidth for n message headers is consumed. In situations where a client sends a number of oneway or datagram invocations, the additional overhead can be considerable.

To avoid the overhead of sending many small messages, you can send oneway and datagram invocations in a batch: instead of being sent as a separate message, a batch invocation is placed into a client-side buffer by the Ice run time. Successive batch invocations are added to the buffer and accumulated on the client side until the client explicitly flushes them. The relevant APIs are part of the proxy interface:

```

namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
            public:

```

```
    Ice::ObjectPrx ice_batchOneway() const;
    Ice::ObjectPrx ice_batchDatagram() const;
    // ...
};

}

}
```

The `ice_batchOneway` and `ice_batchDatagram` methods convert a proxy to a batch proxy. Once you obtain a batch proxy, messages sent via that proxy are buffered in the client-side run time instead of being sent immediately. Once the client has invoked one or more operations on batch proxies, it can explicitly flush the batched invocations by calling `Communicator::flushBatchRequests`:

```
module Ice {
    local interface Communicator {
        void flushBatchRequests();
        // ...
    };
}
```

This causes the batched messages to be sent “in bulk”, preceded by a single message header. On the server side, batched messages are dispatched by a single thread, in the order in which they were written into the batch. This means that messages from a single batch cannot appear to be reordered in the server. Moreover, either all messages in a batch are delivered or none of them. (This is true even for batched datagrams.)

For batched datagram invocations, you need to keep in mind that, if the data for the invocations in a batch substantially exceeds the PDU size of the network, it becomes increasingly likely for an individual UDP packet to get lost due to fragmentation. In turn, loss of even a single packet causes the entire batch to be lost. For this reason, batched datagram invocations are most suitable for simple interfaces with a number of operations that each set an attribute of the target object (or interfaces with similar semantics). (Batched oneway invocations do not suffer from this risk because they are sent over stream-oriented transports, so individual packets cannot be lost.)

Batched invocations are more efficient if you also enable compression for the transport: many isolated and small messages are unlikely to compress well, whereas batched messages are likely to provide better compression because the compression algorithm has more data to work with.⁶

16.13 Testing Proxies for Dispatch Type

The proxy interface offers a number of operations that allow you test the dispatch mode of a proxy. (The Java version of these methods is analogous, so we do not show it here.)

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
            public:
                bool ice_isTwoway() const;
                bool ice_isOneway() const;
                bool ice_isDatagram() const;
                bool ice_isBatchOneway() const;
                bool ice_isBatchDatagram() const;
                // ...
            };
        };
    }
}
```

These operations allow you to test the dispatch mode of an individual proxy.

16.14 The Ice::Logger Interface

Depending on the setting of various properties (see Chapter 14), the Ice run time produces trace, warning, or error messages. These messages are written via the `Ice::Logger` interface:

```
module Ice {
    local interface Logger {
        void trace(string category, string message);
        void warning(string message);
        void error(string message);
    };
};
```

-
6. Regardless of whether you used batched messages or not, you should enable compression only on lower-speed links. For high-speed LAN connections, the CPU time spent doing the compression and decompression is typically longer than the time it takes to just transmit the uncompressed data.

A default logger is instantiated when you create a communicator. The default logger logs to the standard error output. The `trace` operation accepts a `category` parameter in addition to the error message; this allows you to separate trace output from different subsystems by sending the output through a filter.

You can set and get the logger that is attached to a communicator:

```
module Ice {  
    local interface Communicator {  
        void setLogger(Logger log);  
        Logger getLogger();  
    };  
};
```

The `setLogger` operation installs a different logger for a communicator. The Ice run time caches the logger that is set for a communicator. This means that you should set the logger once, after you create the communicator and not change it thereafter. (If you change the logger later, some messages may appear via the old logger and some messages via the new logger.) The `getLogger` operation returns the current logger.

Changing the `Logger` object that is attached to a communicator allows you to integrate Ice messages into your own message handling system. For example, for a complex application, you might have an already existing logging framework. To integrate Ice messages into that framework, you can create your own `Logger` implementation that logs messages to the existing framework.

When you destroy a communicator, its logger is *not* destroyed. This means that you can safely use a logger even beyond the lifetime of its communicator.

For convenience, Ice provides two platform-specific logger implementations: one that logs its messages to the Unix `syslog` facility, and another that uses the Windows event log. You can activate the `syslog` implementation by setting the `Ice.UseSyslog` property, and the Windows event log implementation by setting the `Ice.UseEventLog` property. Both implementations use the value of the `Ice.ProgramName` property to identify the application to the log subsystem.

The `syslog` logger implementation is available in both C++ and Java, whereas the Windows event log implementation is only available in C++. See Section 10.3.2 for additional information on using these logger implementations in C++.

16.15 The Ice::Stats Interface

The Ice run time reports bytes sent and received over the wire on every operation invocation via the `Ice::Stats` interface:

```
module Ice {
    local interface Stats {
        void bytesSent(string protocol, int num);
        void bytesReceived(string protocol, int num);
    };

    local interface Communicator {
        setStats(Stats st);
        // ...
    };
};
```

The Ice run time calls `bytesReceived` as it reads from the network and `bytesSent` as it writes to the network. A very simple implementation of the `Stats` interface could look like the following:

```
class MyStats : public virtual Ice::Stats {
public:
    virtual void bytesSent(const string &prot, Ice::Int num)
    {
        cerr << prot << ": sent " << num << "bytes" << endl;
    }

    virtual void bytesReceived(const string &prot, Ice::Int)
    {
        cerr << prot << ": received " << num << "bytes" << endl;
    }
};
```

To register your implementation, you must instantiate the `MyStats` object and call `setStats` on the communicator:

```
Ice::StatsPtr stats = new MyStats;
communicator->setStats(stats);
```

You can install a `Stats` object on either the client or the server side (or both). Here is some example output produced by installing a `MyStats` object in a simple server:

```
tcp: received 14 bytes
tcp: received 32 bytes
tcp: sent 26 bytes
tcp: received 14 bytes
tcp: received 33 bytes
tcp: sent 25 bytes
...
```

In practice, your Stats implementation will probably be a bit more sophisticated: for example, the object can accumulate statistics in member variables and make the accumulated statistics available via member functions, instead of simply printing everything to the standard error output.

16.16 Location Transparency

One of the useful features of the Ice run time is that it is *location transparent*: the client does not need to know where the implementation of an Ice object resides; an invocation on an object automatically is directed to the correct target, whether the object is implemented in the local address space, in another address space on the same machine, or in another address space on a remote machine. Location transparency is important because it allows us to change the location of an object implementation without breaking client programs and, by using IcePack (see Chapter 21), addressing information such as domain names and port numbers can be externalized so they do not appear in stringified proxies.

For invocations that cross address space boundaries (or more accurately, cross communicator boundaries), the Ice run time dispatches requests via the appropriate transport. However, for invocations that are via proxies for which the proxies and the servants that process the invocation share the same communicator (so-called *collocated* invocations), the Ice run, by default, does not send the invocation via the transport specified in the proxy. Instead, collocated invocations are short-cut inside the Ice run time and dispatched directly.⁷

The reason for this is efficiency: if collocated invocations were sent via TCP/IP, for example, invocations would still be sent via the operating system kernel (using the back plane instead of a network) and would incur the full cost of creating TCP/IP connections, marshaling requests into packets, trapping in and

7. Note that if the proxy and the servant do not use the same communicator, the invocation is *not* collocated, even though caller and callee are in the same address space.

out of the kernel, and so on. By optimizing collocated requests, much of this overhead can be avoided, so collocated invocations are almost as fast as a local function call.

For efficiency reasons, collocated invocations are not completely location transparent, that is, a collocated call has semantics that differ in some ways from calls that cross address-space boundaries. Specifically, collocated invocations differ from ordinary invocations in the following respects:

- Collocated invocations are dispatched in the calling thread instead of being dispatched by a separate thread taken from the server's thread pool.
- The object adapter holding state is ignored: collocated invocations proceed normally even if the target object's adapter is in the holding state.
- For collocated invocations, classes and exceptions are never sliced. Instead, the receiver always receives a class or exception as the derived type that was sent by the sender.
- If a collocated invocation throws an exception that is not in an operation's exception specification, the original exception is raised in the client instead of `UnknownUserException`. (This applies to the C++ mapping only.)
- Asynchronous method invocation (AMI) and asynchronous method dispatch (AMD) cannot be used with collocated invocations.
- Class factories are ignored for collocated invocations.
- Timeouts on invocations are ignored.

In practice, these differences rarely matter. The most likely cause of surprises with collocated invocations is dispatch in the calling thread, that is, a collocated invocation behaves like a local, synchronous procedure call. This can cause problems if, for example, the calling thread acquires a lock that an operation implementation tries to acquire as well: unless you use recursive mutexes (see Chapter 15), this will cause deadlock.

16.17 A Comparison of the Ice and CORBA Run Time

The CORBA equivalent of the server-side functionality of Ice is the Portable Object Adapter (POA). The most striking difference between Ice and the POA is the simplicity of the Ice APIs: Ice is just as fully featured as the POA but achieves this functionality with much simpler interfaces and far fewer operations. Here are a few of the more notable differences:

- Ice object adapters are not hierarchical, whereas POAs are arranged into a hierarchy. It is unclear why CORBA chose to put its adapters into a hierarchy. The hierarchy complicates the POA interfaces but the feature does not provide any apparent benefit: the inheritance of object adapters has no meaning. In particular, POA policies are *not* inherited from the parent adapter. POA hierarchies can be used to control the order of destruction of POAs. However, it is simpler and easier to explicitly destroy adapters in the correct order, as is done in Ice.
- The POA uses a complex policy mechanism to control the behavior of object adapters. The policies can be combined in numerous ways, despite the fact that most combinations do not make sense. This not only is a frequent source of programming errors, but also complicates the API with additional exception semantics for many of its operations.
- The CORBA run time does not provide access to the ORB object (the equivalent of the Ice communicator) during method dispatch. Yet, access to the ORB object is frequently necessary inside operation implementations. As a result, programmers are forced to keep the ORB object in global variable and, if multiple ORBs are used, there is no way to identify the correct ORB for a particular request. The Ice run time eliminates these problems by always providing access to the communicator via the adapter that is passed as part of the `Current` object.
- The POA attaches implementation techniques to object adapters. For example, an object adapter that uses a servant locator cannot also use an active servant map.

The POA also distinguishes between servant locators (which are similar to Ice servant locators) and servant activators (which work like the incrementally initializing servant locator in Section 16.7.1). Yet, there is no need to distinguish the two concepts: a servant activator is simply a special case of a servant locator that can be implemented trivially.

Similarly, default servants must be registered with a POA by making a special API call when, in fact, there is no need to cater for default servants as a separate concept. As shown in Section 16.7.2, with Ice, you can use a trivial servant locator to achieve the same effect.

- The POA uses separate POA manager objects to control adapter states. This not only complicates the APIs considerably, it also makes it possible to combine hierarchies of object adapters with groupings of POA managers in meaningless ways, leading to undefined behavior. Ice does not use separate

objects to control adapter states and so eliminates the associated complexities without loss of functionality.

- The POA interfaces have the notion of a default Root POA that is used unless the programmer overrides it explicitly. This misfeature is a frequent source of errors, due to inappropriate policies that were chosen for the Root POA.
(Users of CORBA will probably have been bitten by implicit activation of objects on the Root POA, instead of the intended POA.)
- The POA provides the concept of implicit activation as well as the notion of system-generated object identities as part of the object adapter policies. This design is a frequent source of programming errors because it leads to many implicit activities behind the scenes, some of them with surprising side effects.
(It is sad to see that all this complexity was added to avoid a single line of code during object activation.) Ice does not provide a notion of implicit activation or implicit generation of object identities. Instead, servants are given an identity explicitly and are activated explicitly, which avoids the complexity and confusion.
- CORBA has no notion of datagrams, or of batched invocations, both of which can provide substantial performance gains.
- CORBA provides no standardized way to integrate ORB messages into existing logging frameworks and does not provide access to network statistics.

In summary, the Ice run time provides all the functionality of the POA (and more) with an API that is a fraction in size. By cleanly separating orthogonal concepts and by providing a minimal but sufficient API, Ice not only provides a simpler and easier-to-use API, but also results in binaries that are smaller in size. This not only reduces the memory requirements of Ice binaries, but also contributes to better performance due to reduced working set size.

16.18 Summary

In this chapter, we explored the server-side run time in detail. Communicators are the main handle to the Ice run time. They provide access to a number of run time resources and allow you to control the life cycle of a server. Object adapters provide a mapping between abstract Ice objects and concrete servants. Various implementation techniques are at your disposal to control the trade-off between performance and scalability; in particular, servant locators are a central mecha-

nism that permits you to choose an implementation technique that matches the requirements of your application.

Ice provides both oneway and datagram invocations. These provide performance gains in situations where an application needs to provide numerous stateless updates. Batching such invocations permits you to increase performance even further.

The Ice logging mechanism is user extensible, so you can integrate Ice message into arbitrary logging frameworks, and the `Ice::Stats` interface permits you to collect statistics for network bandwidth consumption.

Finally, even though Ice is location transparent, in the interest of efficiency, collocated invocations do not behave in all respects like remote invocations. You need to be aware of these differences, especially for applications that are sensitive to thread context.

Chapter 17

Asynchronous Programming

17.1 Chapter Overview

This chapter describes the asynchronous programming facilities in Ice. Section 17.2 gives a brief overview of the capabilities and demonstrates how to modify Slice definitions to enable asynchronous support in language mappings. The client-side facilities are presented in Section 17.3 and are followed by a discussion of the server-side facilities in Section 17.4.

17.2 Introduction

Modern middleware technologies attempt to ease the programmer's transition to distributed application development by making remote invocations as easy to use as traditional method calls: a method is invoked on an object and, when the method completes, the results are returned or an exception is raised. Of course, in a distributed system the object's implementation may reside on another host, and consequently there are some semantic differences that the programmer must be aware of, such as the overhead of remote invocations and the possibility of network-related errors. Despite those issues, the programmer's experience with object-oriented programming is still relevant, and this *synchronous* programming

model, in which the calling thread is blocked until the operation returns, is familiar and easily understood.

Ice is inherently an asynchronous middleware platform that simulates synchronous behavior for the benefit of applications (and their programmers). When an Ice application makes a synchronous two-way invocation on a proxy for a remote object, the operation's parameters are marshaled into a message that is written to a transport, and the calling thread is blocked in order to simulate a synchronous method call. Meanwhile, the Ice run-time operates in the background, processing messages until the desired reply is received and the calling thread can be unblocked to unmarshal the results.

There are many cases, however, in which the blocking nature of synchronous programming is too restrictive. For example, the application may have useful work it can do while it awaits the response to a remote invocation; using a synchronous invocation in this case forces the application to either postpone the work until the response is received, or perform this work in a separate thread. When neither of these alternatives are acceptable, the asynchronous facilities provided with Ice are an effective solution for improving performance and scalability, or simplifying complex application tasks.

17.2.1 Asynchronous Method Invocation

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. Using AMI, a remote invocation does not block the calling thread while the Ice run-time awaits the reply. Instead, the calling thread can continue its activities, and the application is notified by the Ice run-time when the reply eventually arrives. Notification occurs via a callback to an application-supplied programming-language object¹. AMI is described in detail in Section 17.3.

17.2.2 Asynchronous Method Dispatch

The number of simultaneous synchronous requests a server is capable of supporting is limited by the size of the Ice run-time's server thread pool (see Section 15.3). If all of the threads are busy dispatching long-running operations,

1. Polling for a response is not supported by the Ice run-time, but it can be implemented easily by the application if desired.

then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server sends a response explicitly using a callback object provided by the Ice run-time.

In practical terms, an AMD operation typically queues the request data (i.e., the callback object and operation arguments) for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

See Section 17.4 for more information on AMD.

17.2.3 Controlling Code Generation using Metadata

A programmer indicates a desire to use an asynchronous model (AMI, AMD, or both) by annotating Slice definitions with metadata (Section 4.17). The programmer can specify this metadata at two levels: for an interface or class, or for an individual operation. If specified for an interface or class, then asynchronous support is generated for all of its operations. Alternatively, if asynchronous support is needed only for certain operations, then the generated code can be minimized by specifying the metadata only for those operations that require it.

Synchronous invocation methods are always generated in a proxy; specifying AMI metadata merely adds asynchronous invocation methods. In contrast, specifying AMD metadata causes the synchronous dispatch methods to be *replaced* with their asynchronous counterparts. This semantic difference between AMI and AMD is ultimately practical: it is beneficial to provide a client with synchronous and asynchronous versions of an invocation method, but doing the equivalent in a server would require the programmer to implement both versions of the dispatch method, which has no tangible benefits and several potential pitfalls.

Consider the following Slice definitions:

```
["ami"] interface I {
    bool isValid();
    float computeRate();
};

interface J {
    ["amd"] void startProcess();
    ["ami", "amd"] int endProcess();
};
```

In this example, all proxy methods of interface I are generated with support for synchronous and asynchronous invocations. In interface J, the `startProcess` operation uses asynchronous dispatch, and the `endProcess` operation supports asynchronous invocation and dispatch.

Specifying metadata at the operation level, rather than at the interface or class level, not only minimizes the amount of generated code, but more importantly, it minimizes complexity. Although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations for which it provides a particular advantage, while using the simpler synchronous model for the rest.

17.2.4 Transparency

The use of an asynchronous model does not affect what is sent “on the wire.” Specifically, the invocation model used by a client is transparent to the server, and the dispatch model used by a server is transparent to the client. Therefore, a server has no way to distinguish a client’s synchronous invocation from an asynchronous invocation, and a client has no way to distinguish a server’s synchronous reply from an asynchronous reply.

17.3 Using AMI

In this section, we describe the Ice implementation of AMI and how to use it. We begin by discussing a way to (partially) simulate AMI using oneway invocations. This is not a technique that we recommend, but it is an informative exercise that highlights the benefits of AMI and illustrates how it works. Next, we explain the AMI mapping and illustrate its use with examples.

17.3.1 Simulating AMI using Oneways

As we discussed at the beginning of the chapter, synchronous invocations are not appropriate for certain types of applications. For example, an application with a graphical user interface typically must avoid blocking the window system's event dispatch thread because blocking makes the application unresponsive to user commands. In this situation, making a synchronous remote invocation is asking for trouble.

The application could avoid this situation using oneway invocations (see page 16), which by definition cannot return a value or have any out parameters. Since the Ice run-time does not expect a reply, the invocation blocks only as long as it takes to marshal and copy the message into the local transport buffer. However, the use of oneway invocations may require unacceptable changes to the interface definitions. For example, a twoway invocation that returns results or raises user exceptions must be converted into at least two operations: one for the client to invoke with oneway semantics that contains only in parameters, and one (or more) for the server to invoke to notify the client of the results.

To illustrate these changes, suppose that we have the following Slice definition:

```
interface I {  
    int op(string s, out long l);  
};
```

In its current form, the operation `op` is not suitable for a oneway invocation because it has an `out` parameter and a non-void return type. In order to accommodate a oneway invocation of `op`, we can change the Slice definitions as shown below:

```
interface ICallback {  
    void opResults(int result, long l);  
};  
  
interface I {  
    void op(ICallback* cb, string s);  
};
```

We made several modifications to the original definition:

- We added interface `ICallback`, containing an operation `opResults` whose arguments represent the results of the original twoway operation. The server invokes this operation to notify the client of the completion of the operation.

- We modified `I::op` to be compliant with oneway semantics: it now has a void return type, and takes only in parameters.
- We added a parameter to `I::op` that allows the client to supply a proxy for its callback object.

As you can see, we have made significant changes to our interface definitions to accommodate the implementation requirements of the client. One ramification of these changes is that the client must now also be a server, because it must create an instance of `ICallback` and register it with an object adapter in order to receive notifications of completed operations.

A more severe ramification, however, is the impact these changes have on the type system, and therefore on the server. Whether a client invokes an operation synchronously or asynchronously should be irrelevant to the server; this is an artifact of behavior that should have no impact on the type system. By changing the type system as shown above, we have tightly coupled the server to the client, and eliminated the ability for `op` to be invoked synchronously.

To make matters even worse, consider what would happen if `op` could raise user exceptions. In this case, `ICallback` would have to be expanded with additional operations that allow the server to notify the client of the occurrence of each exception. Since exceptions cannot be used as parameter or member types in Slice, this quickly becomes a difficult endeavor, and the results are likely to be equally difficult to use.

At this point, you will hopefully agree that this technique is flawed in many ways, so why do we bother describing it in such detail? The reason is that the Ice implementation of AMI uses a strategy similar to the one described above, with several important differences:

1. No changes to the type system are required in order to use AMI. The on-the-wire representation of the data is identical, therefore synchronous and asynchronous clients and servers can coexist in the same system, using the same operations.
2. The AMI solution accommodates exceptions in a reasonable way.
3. Using AMI does not require the client to also be a server.

17.3.2 Language Mappings

An operation for which the AMI metadata has been specified supports both synchronous and asynchronous invocation models. In addition to the proxy method for synchronous invocation, the code generator creates a proxy method for

asynchronous invocation, plus a supporting callback class. The generated code uses a pattern similar to the Slice modifications we made to the example in Section 17.3.1: the out parameters and return value are removed, leaving only in parameters for the invocation; the application supplies a callback object that is invoked with the results of the operation. In this case, however, the callback object is a purely local entity that is invoked by the Ice run-time in the client. The name of the callback class is constructed so that it cannot conflict with a user-defined Slice identifier.

C++ Mapping

The C++ code generator emits the following code for each AMI operation:

1. An abstract callback class used by the Ice run-time to notify the application about the completion of an operation. The name of the class is formed using the pattern `AMI_class_op`. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Two methods are provided:

```
void ice_response(<params>);
```

Indicates that the operation completed successfully. The parameters represent the return value and out parameters of the operation. If the operation has a non-void return type, then the first parameter of the `ice_response` method supplies the return value of the operation. Any out parameters present in the operation follow the return value, in the order of declaration.

```
void ice_exception(const Ice::Exception &);
```

Indicates that a local or user exception was raised.

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method has a `void` return type. The first parameter is a smart pointer to an instance of the callback class described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
    ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```

class AMI_I_foo : public ... {
public:
    virtual void ice_response(Ice::Int, Ice::Long) = 0;
    virtual void ice_exception(const Ice::Exception &) = 0;
};

typedef IceUtil::Handle<AMI_I_foo> AMI_I_fooPtr;

```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

```
void foo_async(const AMI_I_fooPtr &, Ice::Short);
```

Java Mapping

The Java code generator emits the following code for each AMI operation:

1. An abstract callback class used by the Ice run-time to notify the application about the completion of an operation. The name of the class is formed using the pattern `AMI_class_op`. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Three methods are provided:

```
public void ice_response(<params>);
```

Indicates that the operation completed successfully. The parameters represent the return value and out parameters of the operation. If the operation has a non-void return type, then the first parameter of the `ice_response` method supplies the return value of the operation. Any out parameters present in the operation follow the return value, in the order of declaration.

```
public void ice_exception(Ice.LocalException ex);
```

Indicates that a local exception was raised.

```
public void ice_exception(Ice.UserException ex);
```

Indicates that a user exception was raised.

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method has a `void` return type. The first parameter is a reference to an instance of the callback class described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
    ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```
public abstract class AMI_I_foo extends ... {
    public abstract void ice_response(int __ret, long l);
    public abstract void ice_exception(Ice.LocalException ex);
    public abstract void ice_exception(Ice.UserException ex);
}
```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

```
public void foo_async(AMI_I_foo __cb, short s);
```

17.3.3 Example

To demonstrate the use of AMI in Ice, let us define the Slice interface for a simple computational engine:

```
sequence<float> Row;
sequence<Row> Grid;

exception RangeError {};

interface Model {
    ["ami"] Grid interpolate(Grid data, float factor)
        throws RangeError;
};
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way. In the sections below, we present C++ and Java clients that invoke `interpolate` using AMI.

C++ Client

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

```
class AMI_Model_interpolateI : public AMI_Model_interpolate {
public:
    virtual void ice_response(const Grid & result)
    {
        cout << "received the grid" << endl;
```

```

        // ... postprocessing ...
    }

virtual void ice_exception(const Ice::Exception & ex)
{
    try {
        ex.ice_throw();
    } catch (const RangeError & e) {
        cerr << "interpolate failed: range error" << endl;
    } catch (const Ice::LocalException & e) {
        cerr << "interpolate failed: " << e << endl;
    }
}
};

```

The implementation of `ice_response` reports a successful result, and `ice_exception` displays a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

```

ModelPrx model = ...;
AMI_Model_interpolatePtr cb = new AMI_Model_interpolateI;
Grid grid;
initializeGrid(grid);
model->interpolate_async(cb, grid, 0.5);

```

After obtaining a proxy for a `Model` object, the client instantiates a callback object, initializes a grid and invokes the asynchronous version of `interpolate`. When the Ice run-time receives the response to this request, it invokes the callback object supplied by the client.

Java Client

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

```

class AMI_Model_interpolateI extends AMI_Model_interpolate {
    public void ice_response(float[][] result)
    {
        System.out.println("received the grid");
        // ... postprocessing ...
    }

    public void ice_exception(Ice.UserException ex)
    {
        assert(ex instanceof RangeError);
        System.err.println("interpolate failed: range error");
    }
}

```

```
}

public void ice_exception(Ice.LocalException ex)
{
    System.err.println("interpolate failed: " + ex);
}
}
```

The implementation of `ice_response` reports a successful result, and the `ice_exception` methods display a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

```
ModelPrx model = ...;
AMI_Model_interpolate cb = new AMI_Model_interpolateI();
float[][][] grid = ...;
initializeGrid(grid);
model.interpolate_async(cb, grid, 0.5);
```

After obtaining a proxy for a `Model` object, the client instantiates a callback object, initializes a grid and invokes the asynchronous version of `interpolate`. When the Ice run-time receives the response to this request, it invokes the callback object supplied by the client.

17.3.4 Concurrency Issues

Support for asynchronous invocations in Ice is enabled by the client thread pool (Chapter 15), whose threads are primarily responsible for processing reply messages. It is important to understand the concurrency issues associated with asynchronous invocations:

- A callback object must not be used for multiple simultaneous invocations. An application that needs to aggregate information from multiple replies can create a separate object to which the callback objects delegate.
- Calls to the callback object are made by threads from the Ice run time's client thread pool, therefore synchronization may be necessary if the application might interact with the callback object at the same time as the reply arrives.
- The number of threads in the client thread pool determines the maximum number of simultaneous callbacks possible for asynchronous invocations. The default size of the client thread pool is one, meaning invocations on callback objects are serialized. If the size of the thread pool is increased, the application may require synchronization if the same callback object is used for multiple invocations.

17.4 Using AMD

This section describes the language mappings for AMD and continues the example introduced in Section 17.3.

17.4.1 Language Mappings

As we discussed in Section 17.3.2, the language mappings for AMI continue to allow applications to use the synchronous invocation model if desired: specifying the AMI metadata for an operation leaves the proxy method for synchronous invocation intact, and causes an additional proxy method to be generated in support of asynchronous invocation.

The language mappings for an AMD operation, however, do not allow the implementation to use both dispatch models. Specifying the AMD metadata causes the method for synchronous dispatch to be replaced with a method for asynchronous dispatch.

The asynchronous dispatch method has a signature similar to that of AMI: the return type is `void`, and the arguments consist of a callback object and the operation's in parameters. In AMI the callback object is supplied by the application, but in AMD the callback object is supplied by the Ice run-time and provides methods for returning the operation's results or reporting an exception. The implementation is not required to invoke the callback object before the dispatch method returns; the callback object can be invoked at any time by any thread, but may only be invoked once. The name of the callback class is constructed so that it cannot conflict with a user-defined Slice identifier.

The details for each language mapping are provided below.

C++ Mapping

The C++ code generator emits the following code for each AMD operation:

1. A callback class used by the implementation to notify the Ice run-time about the completion of an operation. The name of this class is formed using the pattern `AMD_class_op`. For example, an operation named `foo` defined in interface `I` results in a class named `AMD_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Several methods are provided:

```
void ice_response(<params>);
```

The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-void return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's out parameters follow the return value, in the order of declaration.

```
void ice_exception(const Ice::Exception &);
```

This version of `ice_exception` allows the server to report a user or local exception.

```
void ice_exception(const std::exception &);
```

This version of `ice_exception` allows the server to report a standard exception.

```
void ice_exception()
```

This version of `ice_exception` allows the server to report an unknown exception.

2. The dispatch method, whose name has the suffix `_async`. This method has a `void` return type. The first parameter is a smart pointer to an instance of the callback class described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
    ["amd"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```
class AMD_I_foo : public ... {
public:
    void ice_response(Ice::Int, Ice::Long);
    void ice_exception(const Ice::Exception &);
    void ice_exception(const std::exception &);
    void ice_exception();
};
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

```
void foo_async(const AMD_I_fooPtr &, Ice::Short);
```

Java Mapping

The Java code generator emits the following code for each AMD operation:

1. A callback interface used by the implementation to notify the Ice run-time about the completion of an operation. The name of this interface is formed using the pattern `AMD_class_op`. For example, an operation named `foo` defined in interface `I` results in an interface named `AMD_I_foo`. The interface is generated in the same scope as the interface or class containing the operation. Two methods are provided:

```
public void ice_response(<params>);
```

The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-void return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's out parameters follow the return value, in the order of declaration.

```
public void ice_exception(java.lang.Exception ex);
```

The `ice_exception` method allows the server to report an exception. With respect to exceptions, there is less compile-time type safety in an AMD implementation because there is no `throws` clause on the dispatch method and any exception type could conceivably be passed to `ice_exception`. However, the Ice run-time validates the exception value using the same semantics as for synchronous dispatch (see Section 4.8.4).

2. The dispatch method, whose name has the suffix `_async`. This method has a `void` return type. The first parameter is a reference to an instance of the callback interface described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
    ["amd"] int foo(short s, out long l);
};
```

The callback interface generated for operation `foo` is shown below:

```
public interface AMD_I_foo {
    void ice_response(int __ret, long l);
    void ice_exception(java.lang.Exception ex);
}
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

```
void foo_async(AMD_I_foo __cb, short s);
```

17.4.2 Exceptions

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (i.e., the thread that receives the invocation), and the response thread (i.e., the thread that sends the response)². Although we recommend that the callback object be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run-time; the application's run-time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Whether raised in a dispatch thread or reported via the callback object, user exceptions are validated as described in Section 4.8.2, and local exceptions may undergo the translation described in Section 4.8.4.

17.4.3 Example

In this section, we continue the example we started in Section 17.3.3, but first we must modify the operation to add AMD metadata:

```
sequence<float> Row;
sequence<Row> Grid;

exception RangeError {};
```

2. These are not necessarily two different threads: the response can also be sent from the dispatch thread if desired.

```
interface Model {
    ["ami", "amd"] Grid interpolate(Grid data, float factor)
        throws RangeError;
};
```

The sections below provide implementations of the `Model` interface in C++ and Java.

C++ Servant

Our servant class derives from `Model` and supplies a definition for the `interpolate_async` method:

```
class ModelI : virtual public Model,
               virtual public IceUtil::Mutex {
public:
    virtual void interpolate_async(
        const AMD_Model_interpolatePtr &,
        const Grid &,
        Ice::Float,
        const Ice::Current &);

private:
    std::list<JobPtr> _jobs;
};
```

The implementation of `interpolate_async` uses synchronization to safely record the callback object and arguments in a `Job` that is added to a queue:

```
void ModelI::interpolate_async(
    const AMD_Model_interpolatePtr & cb,
    const Grid & data,
    Ice::Float factor,
    const Ice::Current & current)
{
    IceUtil::Mutex::Lock sync(*this);
    JobPtr job = new Job(cb, data, factor);
    _jobs.push_back(job);
}
```

After queuing the information, the operation returns control to the Ice run-time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute` to perform the interpolation. `Job` is defined as follows:

```
class Job : public IceUtil::Shared {
public:
    Job(
        const AMD_Model_interpolatePtr &,
        const Grid &,
        Ice::Float);
    void execute();

private:
    bool interpolateGrid();

    AMD_Model_interpolatePtr _cb;
    Grid _grid;
    Ice::Float _factor;
};

typedef IceUtil::Handle<Job> JobPtr;
```

The implementation of `execute` uses `interpolateGrid` (not shown) to perform the computational work:

```
Job::Job(
    const AMD_Model_interpolatePtr & cb,
    const Grid & grid,
    Ice::Float factor) :
    _cb(cb), _grid(grid), _factor(factor)
{
}

void Job::execute()
{
    if(!interpolateGrid()) {
        _cb->ice_exception(RangeError());
        return;
    }
    _cb->ice_response(_grid);
}
```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

Java Servant

Our servant class derives from `_ModelDisp` and supplies a definition for the `interpolate_async` method that creates a `Job` to hold the callback object and arguments, and adds the `Job` to a queue. The method is synchronized to guard access to the queue:

```
public final class ModelI extends _ModelDisp {
    synchronized public void interpolate_async(
        AMD_Model_interpolate cb,
        float[][] data,
        float factor,
        Ice.Current current)
            throws RangeError
    {
        _jobs.add(new Job(cb, data, factor));
    }

    java.util.LinkedList _jobs = new java.util.LinkedList();
}
```

After queuing the information, the operation returns control to the Ice run-time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

```
class Job {
    Job(AMD_Model_interpolate cb,
        float[][] grid,
        float factor)
    {
        _cb = cb;
        _grid = grid;
        _factor = factor;
    }

    void execute()
    {
        if(!interpolateGrid()) {
            _cb.ice_exception(new RangeError());
            return;
        }
        _cb.ice_response(_grid);
    }

    private boolean interpolateGrid() {
```

```
// ...
}

private AMD_Model_interpolate _cb;
private float[][] _grid;
private float _factor;
}
```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

17.5 Summary

Synchronous remote invocations are a natural extension of local method calls that leverage the programmer's experience with object-oriented programming and ease the learning curve for programmers who are new to distributed application development. However, the blocking nature of synchronous invocations makes some application tasks more difficult, or even impossible, therefore Ice provides a straightforward interface to its asynchronous facilities.

Using asynchronous method invocation, a calling thread is able to invoke an operation and regain control immediately, without blocking while the operation is in progress. When the results are received, the Ice run-time notifies the application via a callback.

Similarly, asynchronous method dispatch allows a servant to send the results of an operation at any time, not necessarily within the operation implementation. A servant can improve scalability and conserve thread resources by queuing time-consuming requests for processing at a later time.

Chapter 18

Facets and Versioning

18.1 Introduction

Facets provide a general-purpose mechanism for non-intrusively extending the type system of an application. This is particularly useful for versioning an application. Section 18.2 introduces the facet concept and presents the relevant APIs. Section 18.3 presents a few traditional approaches to versioning and their problems. Sections 18.4 to 18.6 show how to use facets to implement versioning, and Section 18.7 discusses design choices when adding versioning to a system.

18.2 Concept and APIs

Up to this point, we have presented an Ice object as a single conceptual entity, that is, as an object with a single most-derived interface and a single identity, with the object being implemented by a single servant. However, an Ice object is more

correctly viewed as a collection of one or more sub-objects known as *facets*, as shown in Figure 18.1.

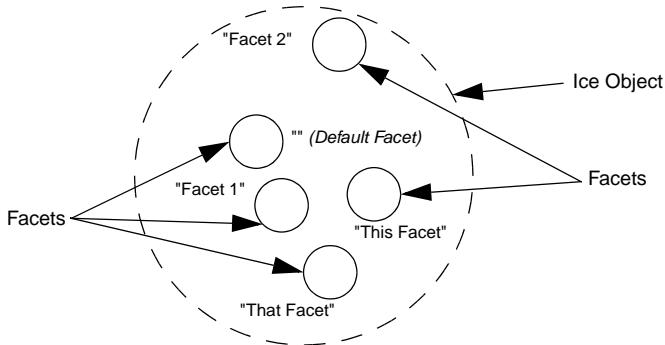


Figure 18.1. An Ice object with five facets sharing a single object identity.

Figure 18.1 shows a single Ice object with five facets. Each facet has a name, known as the *facet name*. Within a single Ice object, all facets must have unique names. Facet names are arbitrary strings that are assigned by the server that implements an Ice object. A facet with an empty facet name is legal and known as the *default facet*. Unless you arrange otherwise, an Ice object has a single default facet; by default, operations that involve Ice objects and servants operate on the default facet.

Note that all the facets of an Ice object share the same single identity, but have different facet names. Recall the definition of `Ice::Current` we saw in Section 16.5 once more:

```
module Ice {
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Nonmutating, \Idempotent };

    local struct Current {
        ObjectAdapter    adapter;
        Identity         id;
        string           facet;
        string           operation;
        OperationMode   mode;
        Context          ctx;
    };
};
```

By definition, if two facets have the same `id` field, they are part of the same Ice object. Also by definition, if two facets have the same `id` field, their `facet` fields have different values.

Even though Ice objects usually consist of just the default facet, it is entirely legal for an Ice object to consist of facets that all have non-empty names (that is, it is legal for an Ice object not to have a default facet).

Each facet has a single most-derived interface. There is no need for the interface types of the facets of an Ice object to be unique. It is legal for two facets of an Ice object to implement the same most-derived interface.

Each facet is implemented by a servant. All the usual implementation techniques for servants are available to implement facets—for example, you can implement a facet using a servant locator. Typically, each facet of an Ice object has a separate servant, although, if two facets of an Ice object have the same type, they can also be implemented by a single servant (for example, using a default servant, as described in Section 16.7.2).

18.2.1 Server-Side Facet Operations

On the server side, the object adapter offers a number of operations to support facets:

```
namespace Ice {
    dictionary<string, Object> FacetMap;

    local interface ObjectAdapter {
        Object* addFacet(Object servant, Identity id, string facet);
        Object* addFacetWithUUID(Object servant, string facet);
        Object removeFacet(Identity id, string facet);
        Object findFacet(Identity id, string facet);

        FacetMap findAllFacets(Identity id);
        FacetMap removeAllFacets(Identity id);
        // ...
    };
}
```

These operations have the same semantics as the corresponding “normal” operations (`add`, `addWithUUID`, `remove`, and `find`), but also accept a facet name. The corresponding “normal” operations are simply convenience operations that supply an empty facet name. For example, `remove(id)` is equivalent to `removeFacet(id, "")`, that is, `remove(id)` operates on the default facet.

`findAllFacets` returns a dictionary of `<facet-name, servant>` pairs that contains all the facets for the given identity.

`removeAllFacets` removes all facets for a given identity from the active servant map, that is, it removes the corresponding Ice object entirely. The operation returns a dictionary of `<facet-name, servant>` pairs that contains all the removed facets.

These operations are sufficient for the server to create Ice objects with any number of facets. For example, assume that we have the following Slice definitions:

```
module Filesystem {  
    // ...  
  
    interface File extends Node {  
        nonmutating Lines read();  
        idempotent void write(Lines text) throws GenericError;  
    };  
};  
  
module FilesystemExtensions {  
    // ...  
  
    class DateTime extends TimeOfDay {  
        // ...  
    };  
  
    struct Times {  
        DateTime createdDate;  
        DateTime accessedDate;  
        DateTime modifiedDate;  
    };  
  
    interface Stat {  
        nonmutating Times getTimes();  
    };  
};
```

Here, we have a `File` interface that provides operations to read and write a file, and a `Stat` interface that provides access to the file creation, access, and modification time. (Note that the `Stat` interface is defined in a different module and could also be defined in a different source file.) If the server wants to create an Ice object that contains a `File` instance as the default facet and a `Stat` instance that provides access to the time details of the file, it could use:

```
// Create a File instance.  
//  
Filesystem::FilePtr file = new FileI();  
  
// Create a Stat instance.  
//  
FilesystemExceptions::DateTimePtr dt  
= new FilesystemExtensions::DateTime();  
FilesystemExtensions::Times times;  
times.createdDate = dt;  
times.accessedDate = dt;  
times.modifiedDate = dt;  
FilesystemExtensions::StatPtr stat = new StatI(times);  
  
// Register the File instance as the default facet.  
//  
Filesystem::FilePrx filePrx = myAdapter->addWithUUID(file);  
  
// Register the Stat instance as a facet with name "Stat".  
//  
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

The first few lines simply create and initialize a `FileI` and `StatI` instance. (The details of this do not matter here.) All the action is in the last two statements:

```
Filesystem::FilePrx filePrx = myAdapter->addWithUUID(file);  
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

This registers the `FileI` instance with the object adapter as usual. (In this case, we let the Ice run time generate a UUID as the object identity.) Because we are calling `addWithUUID` (as opposed to `addFacetWithUUID`), the instance becomes the default facet.

The second line adds a facet to the instance with the facet name `Stat`. Note that we call `ice_getIdentity` on the `File` proxy to pass an object identity to `addFacet`. This guarantees that the two facets share the same object identity.

Note that, in general, it is a good idea to use `ice_getIdentity` to obtain the identity of an existing facet when adding a new facet. That way, it is guaranteed that the facets share the same identity. (If you accidentally pass a different identity to `addFacet`, you will not add a facet to an existing Ice object, but instead register a new Ice object; using `ice_getIdentity` makes this mistake impossible.)

18.2.2 Client-Side Facet Operations

On the client side, which facet a request is addressed to is implicit in the proxy that is used to send the request. For an application that does not use facets, the facet name is always empty so, by default, requests are sent to the default facet.

The client can use a `checkedCast` to obtain a proxy for a particular facet. For example, assume that the client obtains a proxy to a `File` instance as shown in Section 18.2.1. The client can cast between the `File` facet and the `Stat` facet (and back) as follows:

```
// Get a File proxy.  
//  
Filesystem::FilePrx file = ...;  
  
// Get the Stat facet.  
//  
FilesystemExtensions::StatPrx stat  
= FilesystemExtensions::StatPrx::checkedCast(file, "Stat");  
  
// Go back from the Stat facet to the File facet.  
//  
Filesystem::FilePrx file2  
= Filesystem::FilePrx::checkedCast(stat, "");  
  
assert(file2 == file); // The two proxies are identical.
```

This example illustrates that, given any facet of an Ice object, the client can navigate to any other facet by using a `checkedCast` with the facet name.

If an Ice object does not provide the specified facet, `checkedCast` returns null:

```
FilesystemExtensions::StatPrx stat  
= FilesystemExtensions::StatPrx::checkedCast(file, "Stat");  
  
if (!stat) {  
    // No Stat facet on this object, handle error...  
} else {  
    FilesystemExtensions::Times times = stat->getTimes();  
  
    // Use times struct...  
}
```

Note that `checkedCast` also returns a null proxy if a facet exists, but the cast is to the wrong type. For example:

```
// Get a File proxy.  
//  
Filesystem::FilePrx file = ...;  
  
// Cast to the wrong type.  
//  
SomeTypePrx prx = SomeTypePrx::checkedCast(file, "Stat");  
  
assert(!prx); // checkedCast returns a null proxy.
```

If you want to distinguish between non-existence of a facet and the facet being of the incorrect type, you can first obtain the facet as type `Object` and then down-cast to the correct type:

```
// Get a File proxy.  
//  
Filesystem::FilePrx file = ...;  
  
// Get the facet as type Object.  
//  
Ice::ObjectPrx obj = Ice::ObjectPrx::checkedCast(file, "Stat");  
if (!obj) {  
    // No facet with name "Stat" on this Ice object.  
} else {  
    FilesystemExtensions::StatPrx stat =  
        FilesystemExtensions::StatPrx::checkedCast(file);  
    if (!stat) {  
        // There is a facet with name "Stat", but it is not  
        // of type FilesystemExtensions::Stat.  
    } else {  
        // Use stat...  
    }  
}
```

This last example also illustrates that

```
StatPrx::checkedCast(prx, "")
```

is *not* the same as

```
StatPrx::checkedCast(prx)
```

The first version explicitly requests a cast to the default facet. This means that the Ice run time first looks for a facet with the empty name and then attempts to down-cast that facet (if it exists) to the type `Stat`.

The second version requests a down-cast that *preserves* whatever facet is currently effective in the proxy. For example, if the `prx` proxy currently holds the

facet name “Joe”, then (if `prx` points at an object of type `Stat`) the run time returns a proxy of type `StatPrx` that also stores the facet name “Joe”.

It follows that, to navigate between facets, you must always use the two-argument version of `checkedCast`, whereas, to down-cast to another type while preserving the facet name, you must always use the single-argument version of `checkedCast`.

You can always check what the current facet of a proxy is by calling `ice_getFacet`:

```
Ice::ObjectPrx obj = ...;

cout << obj->ice_getFacet() << endl; // Print facet name
```

This prints the facet name. (For the default facet, `ice_getFacet` returns the empty string.)

18.2.3 Exception Semantics

As we pointed out on page 83, `ObjectNotExistException` and `FacetNotExistException` have the following semantics:

- `ObjectNotExistException`

This exception is raised only if no facets exist at all for a given object identity.

- `FacetNotExistException`

This exception is raised only if at least one facet exists for a given object identity, but not the specific facet that is the target of an operation invocation.

If you are using servant locators (see Section 16.6) or default servants (Section 16.7.2), you must take care to preserve these semantics. (In particular, if you return null from a servant locator’s `locate` operation, this appears to the client as an `ObjectNotExistException`. If the object identity for a request is known (that is, there is at least one facet with that identity), but no facet with the specified name exists, you must explicitly throw a `FacetNotExistException` from `activate` instead of simply returning null.)

18.3 The Versioning Problem

Once you have developed and deployed a distributed application, and once the application has been in use for some time, it is likely that you will want to make some changes to the application. For example, you may want to add new function-

ability to a later version of the application, or you may want to change some existing aspect of the application. Of course, ideally, such changes are accomplished without breaking already deployed software, that is, the changes should be backward compatible. Evolving an application in this way is generally known as *versioning*.

Versioning is an aspect that previous middleware technologies have addressed only poorly (if at all). One of the purposes of facets is to allow you to cleanly create new versions of an application without compromising compatibility with older, already deployed versions.

18.3.1 An Attempt at Versioning

Suppose that we have deployed our file system application and want to add extra functionality to a new version. Specifically, let us assume that the original version (version 1) only provides the basic functionality to use files, but does not provide extra information, such as the modification date or the file size. The question is then, how can we upgrade the existing application with this new functionality? Here is a small excerpt of the original (version 1) Slice definitions once more:

```
// Version 1

module Filesystem {
    // ...

    interface File extends Node {
        nonmutating Lines read();
        idempotent void write(Lines text) throws GenericError;
    };
}
```

Your first attempt at upgrading the application might look as follows:

```
// Version 2

module Filesystem {
    // ...

    class DateTime extends TimeOfDay { // New in version 2
        // ...
    };

    struct Times {                                // New in version 2
        DateTime createdDate;
    }
}
```

```
        DateTime accessedDate;
        DateTime modifiedDate;
    };

interface File extends Node {
    nonmutating Lines read();
    idempotent void write(Lines text) throws GenericError;

    nonmutating Times getTimes(); // New in version 2
};

};
```

At first glance, this seems reasonable: the version 2 definition does not change anything that was present in version 1; instead it only adds two new types and adds an operation to the `File` interface. This should mean that version 1 clients will continue to work with both version 1 and version 2 `File` objects because version 1 clients do not know about the `getTimes` operation and therefore will not call it; version 2 clients, on the other hand, can take advantage of the new functionality.

Unfortunately, this approach is severely flawed for the following reasons:

- The tacit assumption built into this approach is that somehow, all existing version 1 servers will be upgraded to version 2 servers more or less atomically. In other words, once there are deployed version 2 clients, it is assumed that *all* `File` objects will support the `getTimes` operation.

Unfortunately, making this assumption is dangerous because, in real-world scenarios, it is difficult to ensure that all parts of a system are upgraded at the same time. The danger, of course, is that a version 2 client somehow ends up talking to a version 1 `File` object and then gets a big surprise in the form of an `OperationNotExistException` when it calls `getTimes`. What is worse, there is no way for a version 2 client to somehow “ask” a `File` object whether it is a version 1 or version 2 object because, in both versions, the objects have the same type (`Filesystem::File`).

- If you implement the above approach and try it out, you will find that a version 1 client can indeed transparently use a version 2 `File` object. However, the fact that this works is purely coincidental and merely an artifact of how the Ice protocol invokes an operation: currently, the Ice protocol (see Chapter 19) indicates the operation to be invoked by sending the operation name as a string on the wire. However, a future version of the protocol might change this. For example, a future version of the protocol might use a hash value instead of a string to indicate an operation name. In that case, adding a

new operation to an interface can change the hash value of an existing operation, with the result that version 1 clients cannot use a version 2 `File` object at all.

The importance of the second point cannot be overstated, so here it is in clear text once more:

The Ice type system requires that the meaning of a type is the same everywhere within a communications domain. It is illegal to have two types with the same name, but different definitions.

Of course, the approach we just examined violates this rule if version 1 and version 2 clients co-exist: to version 1 clients, the type `File` has a different meaning than to version 2 clients, which is illegal.

18.3.2 Versioning by Derivation

Given that the previous approach is illegal, you may decide to upgrade the application as follows instead:

```
module Filesystem {      // Version 1
    // ...

    interface File extends Node {
        nonmutating Lines read();
        idempotent void write(Lines text) throws GenericError;
    };
};

module FilesystemV2 {    // New in version 2
    // ...

    class DateTime extends TimeOfDay {
        // ...
    };

    struct Times {
        DateTime createdDate;
        DateTime accessedDate;
        DateTime modifiedDate;
    };
};
```

```
interface File extends Filesystem::File {  
    nonmutating Times getTimes();  
};  
};
```

The idea is to present the new functionality in an interface that is derived from the version 1 interface. This is clean as far as the type system is concerned. The version 1 types are unchanged and the new functionality is presented via new types that are backward compatible: a version 2 `File` object can be passed where a version 1 `File` object is expected because `FilesystemV2::File` is derived from `Filesystem::File`. Even better, if a version 2 component of the system receives a proxy of formal type `Filesystem::File`, it can determine at run time whether the actual run-time type is `FilesystemV2::File` by attempting a down-cast: if the down-cast succeeds, it is dealing with a version 2 object; if the down-cast fails, it is dealing with a version 1 object.

At this point, you may think that versioning by derivation solves the problem elegantly. Unfortunately, the truth turns out to be a little harsher:

- As the system evolves further, and new versions are added, each new version adds a level of derivation to the inheritance tree. After a few versions, particularly if your application also uses inheritance for its own purposes, the resulting inheritance graph very quickly turns into a complex mess. (This becomes most obvious if the application uses multiple inheritance—after a few versioning steps, the resulting inheritance hierarchy is usually so complex that it exceeds the ability of humans to comprehend it.)
- Real-life versioning requirements are not as simple as adding a new operation to an object. Frequently, versioning requires changes such as adding a field to a structure, adding a parameter to an operation, changing the type of a field or a parameter, renaming an operation, or adding a new exception to an operation. However, versioning by derivation can handle none of these changes.
- Quite often, functionality that is present in an earlier version needs to be removed for a later version (for example, because the older functionality has been supplanted by a different mechanism or turned out to be inappropriate). However, there is no way to *remove* functionality through versioning by derivation. The best you can do is to re-implement a base operation in the derived implementation of an interface and throw an exception. However, the deprecated operation may not have an exception specification, or if it does, the exception specification may not include suitable exception. And, of course, doing this perverts the type system: after all, if an interface has an operation

that throws an exception whenever the operation is invoked, why does the operation exist in the first place?

There are other, more subtle reasons why versioning by derivation is unsuitable in real-life situations. Suffice it to say here that experience has shown the idea to be unworkable: projects that have tried to use this technique for anything but the most trivial versioning requirements have inevitably failed.

18.3.3 Explicit Versioning

Yet another twist on the versioning theme is to explicitly version everything, for example:

```
module Filesystem {  
    // ...  
  
    interface FileV1 extends NodeV1 {  
        nonmutating LinesV1 read();  
        idempotent void write(LinesV1 text) throws GenericErrorV1;  
    };  
  
    class DateTimeV2 extends TimeOfDayV2 {  
        // ...  
    };  
  
    struct TimesV2 {  
        DateTimeV2 createdDate;  
        DateTimeV2 accessedDate;  
        DateTimeV2 modifiedDate;  
    };  
  
    interface FileV2 extends NodeV2 {  
        nonmutating LinesV2 read();  
        idempotent void write(LinesV2 text) throws GenericErrorV2;  
        nonmutating TimesV2 getTimes();  
    };  
};
```

In essence, this approach creates as many separate definitions of each data type, interface, and operation as there are versions. It is easy to see that this approach does not work very well:

- Because, at the time version 1 is produced, it is unknown what might need to change for version 2 and later versions, *everything* has to be tagged with a version number. This very quickly leads to an incomprehensible type system.

- Because every version uses its own set of separate types, there is no type compatibility. For example, a version 2 type cannot be passed where a version 1 type is expected without explicit copying.
- Client code must be written to explicitly deal with each separate version. This pollutes the source code at all points where a remote call is made or a Slice data type is passed; the resulting code quickly becomes incomprehensible.

Other approaches, such as placing the definitions for each version into a separate module (that is, versioning the enclosing module instead of each individual type) do little to mitigate these problems; the type incompatibility issues and the need to explicitly deal with versioning remain.

18.4 Versioning with Facets

A negative aspect of all the approaches in Section 18.3 is that they change the type system in intrusive ways. In turn, this forces unacceptable programming contortions on clients. Facets allow you to solve the versioning problem more elegantly because they do not change an existing type system but extend it instead. We already saw this approach in operation on page 470, where we added date information about a file to our file system application without disturbing any of the existing definitions.

In the most general sense, facets provide a mechanism for implementing multiple interfaces for a single object. The key point is that, to add a new interface to an object, none of the existing definitions have to be touched, so no compatibility issues can arise. More importantly, the decision as to which facet to use is made at run time instead of at compile time. In effect, facets implement a form of late binding and, therefore, are coupled to the type system more loosely than any of the previous approaches.

Used judiciously, facets can handle versioning requirements more elegantly than other mechanisms. Apart from the straight extension of an interface as shown in Section 18.2.1, facets can also be used for more complex changes. For example, if you need to change the parameters of an operation or modify the fields of a structure, you can create a new facet with operations that operate on the changed data types. Quite often, the implementation of a version 2 facet in the server can even re-use much of the version 1 functionality, by delegating some version 2 operations to a version 1 implementation.

18.5 Facet Selection

Given that we have decided to extend an application with facets, we have to deal with the question of how clients select the correct facet. The answer typically involves an explicit selection of a facet sometime during client start-up. For example, in our file system application, clients always begin their interactions with the file system by creating a proxy to the root directory. Let us assume that our versioning requirements have led to version 1 and version 2 definitions of directories as follows:

```
module Filesystem { // Original version
    // ...

    interface Directory extends Node {
        nonmutating NodeSeq list();
        // ...
    };
};

module FilesystemV2 {
    // ...

    enum NodeType { Directory, File };

    class NodeDetails {
        NodeType type;
        string name;
        DateTime createdTime;
        DateTime accessedTime;
        DateTime modifiedTime;
        // ...
    };

    interface Directory extends Filesystem::Node {
        nonmutating NodeDetailsSeq list();
        // ...
    };
};
```

In this case, the semantics of the `list` operation have changed in version 2. A version 1 client uses the following code to obtain a proxy to the root directory:

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base
    = communicator()->stringToProxy("RootDir:default -p 10000");
if (!base)
    throw "Could not create proxy";

// Down-cast the proxy to a Directory proxy
//
Filesystem::DirectoryPrx rootDir
    = Filesystem::DirectoryPrx::checkedCast(base);
if (!rootDir)
    throw "Invalid proxy";
```

For a version 2 client, the bootstrap code is almost identical—instead of down-casting to `Filesystem::Directory`, the client selects the “V2” facet during the down-cast to the type `FilesystemV2::Directory`:

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base
    = communicator()->stringToProxy("RootDir:default -p 10000");
if (!base)
    throw "Could not create proxy";

// Down-cast the proxy to a V2 Directory proxy
//
FilesystemV2::DirectoryPrx rootDir
    = FilesystemV2::DirectoryPrx::checkedCast(base, "V2");
if (!rootDir)
    throw "Invalid proxy";
```

Of course, we can also create a client that can deal with both version 1 and version 2 directories: if the down-cast to version 2 fails, the client is dealing with a version 1 server and can adjust its behavior accordingly.

18.6 Behavioral Versioning

On occasion, versioning requires changes in behavior that are not manifest in the interface of the system. For example, we may have an operation that performs some work, such as:

```
interface Foo {  
    void doSomething();  
};
```

The same operation on the same interface exists in both versions, but the *behavior* of `doSomething` in version 2 differs from that in version 1. The question is, how do we best deal with such behavioral changes?

Of course, one option is to simply create a version 2 facet and to carry that facet alongside the original version 1 facet. For example:

```
module V2 {  
  
    interface Foo {    // V2 facet  
        void doSomething();  
    };  
};
```

This works fine, as far as it goes: a version 2 client asks for the “V2” facet and then calls `doSomething` to get the desired effect. Depending on your circumstances, this approach may be entirely reasonable. However, if there are such behavioral changes on several interfaces, the approach leads to a more complex type system because it duplicates each interface with such a change.

A better alternative can be to create two facets of the same type, but have the implementation of those facets differ. With this approach, both facets are of type `::Foo::doSomething`. However, the implementation of `doSomething` checks which facet was used to invoke the request and adjusts its behavior accordingly:

```
void  
FooI::doSomething(const Ice::Current &c)  
{  
    if(c.facet == "V2") {  
        // Provide version 2 behavior...  
    } else {  
        // Provide version 1 behavior...  
    }  
}
```

This approach avoids creating separate types for the different behaviors, but has the disadvantage that version 1 and version 2 objects are no longer distinguishable to the type system. This can matter if, for example, an operation accepts a `Foo` proxy as a parameter. Let us assume that we also have an interface `FooProcessor` as follows:

```
interface FooProcessor {
    void processFoo(Foo* w);
};
```

If `FooProcessor` also exists as a version 1 and version 2 facet, we must deal with the question of what should happen if a version 1 `Foo` proxy is passed to a version 2 `processFoo` operation because, at the type level, there is nothing to prevent this from happening.

You have two options to deal with this situation:

- Define working semantics for mixed-version invocations. In this case, you must come up with sensible system behavior for all possible combinations of versions.
- If some of the mixed-version combinations are disallowed (such as passing a version 1 `Foo` proxy to a version 2 `processFoo` operation), you can detect the version mismatch in the server by looking at the `Current::facet` member and throwing an exception to indicate a version mismatch. Simultaneously, write your clients to ensure they only pass a permissible version to `processFoo`. Clients can ensure this by checking the facet name of a proxy before passing it to `processFoo` and, if there is a version mismatch, changing either the `Foo` proxy or the `FooProcessor` proxy to a matching facet:

```
FooPrx fooPrx foo = ...;           // Get a Foo...
FooProcessorPrx fooP = ...;         // Get a FooProcessor...

string fooFacet = foo->ice_getFacet();
string fooPFacet = foo->ice_getFacet();
if(fooFacet != fooPFacet)
{
    if(fooPFacet == "V2") {
        error("Cannot pass a V1 Foo to a V2 FooProcessor");
    } else {
        // Upgrade FooProcessor from V1 to V2
        fooP = FooProcessorPrx::checkedCast(fooP, "V2");
        if(!fooP) {
            error("FooProcessor does not have a V2 facet");
        } else {
            fooP->processFoo(foo);
        }
    }
}
```

18.7 Design Considerations

Facets allow you to add versioning to a system, but they are merely a mechanism, not a solution. You still have to make a decision as to how to version something. For example, at some point, you may want to deprecate a previous version's behavior; at that point, you must make a decision how to handle requests for the deprecated version. For behavioral changes, you have to decide whether to use separate interfaces or use facets with the same interface. And, of course, you must have compatibility rules to determine what should happen if, for example, a version 1 object is passed to an operation that implements version 2 behavior. In other words, facets cannot do your thinking for you and are no panacea for the versioning problem.

The biggest advantage of facets is also the biggest drawback: facets delay the decision about the types that are used and their behavior until run time. While this provides a lot of flexibility, it is significantly less type safe than having explicit types that can be statically checked at compile time: if you have a problem relating to incorrect facet selection, the problem will be visible only at run time and, moreover, will be visible only if you actually execute the code that contains the problem, and execute it with just the right data.

Another danger of facets is to abuse them. As an extreme example, here is an interface that provides an arbitrary collection of objects of arbitrary type:

```
interface Collection {};
```

Even though this interface is empty, it can provide access to an unlimited number of objects of arbitrary type in the form of facets. While this example is extreme, it illustrates the design tension that is created by facets: you must decide, for a given versioning problem, how and at what point of the type hierarchy to split off a facet that deals with the changed functionality. The temptation may be to “simply add another facet” and be done with it. However, if you do that, your objects are in danger of being nothing more than loose conglomerates of facets without rhyme or reason, and with little visibility of their relationships in the type system.

In object modeling terms, the relationship among facets is weaker than an *is-a* relationship (because facets are often not type compatible among each other). On the other hand, the relationship among facets is stronger than a *has-a* relationship (because all facets of an Ice object share the same object identity).

It is probably best to treat the relationship of a facet to its Ice object with the same respect as an inheritance relationship: if you were omniscient and could have designed your system for all current and future versions simultaneously,

many of the operations that end up on separate facets would probably have been in the same interface instead. In other words, adding a facet to an Ice object most often implies that the facet has an *is-partly-a* relationship with its Ice object. In particular, if you think about the life cycle of an Ice object and find that, when an Ice object is deleted, all its facets must be deleted, this is a strong indication of a correct design. On the other hand, if you find that, at various times during an Ice object's life cycle, it has a varying number of facets of varying types, that is a good indication that you are using facets incorrectly.

Ultimately, the decision comes down to deciding whether the trade-off of static type safety versus dynamic type safety is worth the convenience and backward compatibility. The answer depends strongly on the design of your system and individual requirements, so we can only give broad advice here. Finally, there will be a point where no amount of facet trickery will get past the point when “yet one more version will be the straw that breaks the camel’s back.” At that point, it is time to stop supporting older versions and to redesign the system.

18.8 Summary

Facets provide a way to extend a type system by loosely coupling new type instances to existing ones. This shifts the type selection process from compile to run time and implements a form of late binding. Due to their loose coupling among each other, facets are better suited to solve the versioning problem than other approaches. However, facets are not a panacea that would solve the versioning problem for free, and careful design is still necessary to come up with versioned systems that remain understandable and maintain consistent semantics.

Chapter 19

The Ice Protocol

19.1 Chapter Overview

The Ice protocol definition consists of three major parts:

- a set of data encoding rules that determine how the various data types are serialized
- a number of message types that are interchanged between client and server, together with rules as to what message is to be sent under what circumstances
- a set of rules that determine how client and server agree on a particular protocol and encoding version

Section 19.2 describes the encoding rules, Section 19.3 describes the various protocol messages, Section 19.4 describes compression, and Section 19.5 explains how the protocol and encoding are versioned and how client and server agree on a common version. (Both encoding and protocol specifications are currently at version 1.0.) Finally, Section 19.6 provides a comparison of the Ice protocol and encoding with those used by CORBA.

19.2 Data Encoding

The key goals of the Ice data encoding are simplicity and efficiency. In keeping with these principles, the encoding does not align primitive types on word boundaries and therefore eliminates the wasted space and additional complexity that alignment requires. The Ice data encoding simply produces a stream of contiguous bytes; data contains no padding bytes and need not be aligned on word boundaries.

Data is always encoded using little-endian byte order for numeric types. (Most machines use a little-endian byte order, so the Ice data encoding is “right” more often than not.) Ice does not use a “receiver makes it right” scheme because of the additional complexity this would introduce. Consider, for example, a chain of receivers that merely forward data along the chain until that data arrives at an ultimate receiver. (Such topologies are common for event distribution services.) The Ice protocol permits all the intermediates to forward the data without requiring it to be unmarshaled: the intermediates can forward requests by simply copying blocks of binary data. With a “receiver makes it right” scheme, the intermediates would have to unmarshal and remarshal the data whenever the byte order of the next receiver in the chain differs from the byte order of the sender, which is inefficient.

Ice requires clients and servers that run on big-endian machines to incur the extra cost of byte swapping data into little-endian layout, but that cost is insignificant compared to the overall cost of sending or receiving a request.

19.2.1 Sizes

Many of the types involved in the data encoding, as well as several protocol message components, have an associated size or count. A size is a non-negative number. Sizes and counts are encoded in one of two ways:

1. If the number of elements is less than 255, the size is encoded as a single `byte` indicating the number of elements.
2. If the number of elements is greater than or equal to 255, the size is encoded as a `byte` with value 255, followed by an `int` indicating the number of elements.

Using this encoding to indicate sizes is significantly cheaper than always using an `int` to store the size, especially when marshaling sequences of short strings: counts of up to 254 require only a single byte instead of four. This comes at the expense of counts greater than 254, which require five bytes instead of four.

However, for sequences or strings of length greater than 254, the extra byte is insignificant.

19.2.2 Encapsulations

An encapsulation is used to contain variable-length data that an intermediate receiver may not be able to decode, but that the receiver can forward to another recipient for eventual decoding. An encapsulation is encoded as if it were the following structure:

```
struct Encapsulation {  
    int size;  
    byte major;  
    byte minor;  
    // [... size - 6 bytes ...]  
};
```

The `size` member specifies the size of the encapsulation in bytes (including the `size`, `major`, and `minor` fields). The `major` and `minor` fields specify the encoding version of the data contained in the encapsulation (see Section 19.5.2). The version information is followed by `size`-6 bytes of encoded data.

All the data in an encapsulation is context-free, that is, nothing inside an encapsulation can refer to anything outside the encapsulation. This property allows encapsulations to be forwarded among address spaces as a blob of data.

Encapsulations can be nested, that is, contain other encapsulations.

An encapsulations can be empty, in which case its byte count is 6.

19.2.3 Slices

Exceptions and classes are subject to slicing if the receiver of a value only partially understands the received value (that is, only has knowledge of a base type, but not of the actual run-time derived type). To allow the receiver of an exception or class to ignore those parts of a value that it does not understand, exception and class values are marshaled as a sequence of slices (one slice for each level of the inheritance hierarchy). A slice is a byte count encoded as a fixed-length four-byte integer, followed by the data for the slice. (The byte count includes the four bytes occupied by the count itself, so an empty slice has a byte count of four and no data.) The receiver of a value can skip over a slice by reading the byte count b , and then discarding the next $b-4$ bytes in the input stream.

A slice can be empty, in which case its byte count is 4.

19.2.4 Basic Types

The basic types are encoded as shown in Table 19.1. Integer types (`short`, `int`, `long`) are represented as two's complement numbers, and floating point types (`float`, `double`) use the IEEE standard formats [6]. All numeric types use a little-endian byte order.

Table 19.1. Encoding for basic types.

Type	Encoding
<code>bool</code>	A single byte with value 1 for <code>true</code> , 0 for <code>false</code>
<code>byte</code>	An uninterpreted byte
<code>short</code>	Two bytes (LSB, MSB)
<code>int</code>	Four bytes (LSB .. MSB)
<code>long</code>	Eight bytes (LSB .. MSB)
<code>float</code>	Four bytes (23-bit fractional mantissa, 8-bit exponent, sign bit)
<code>double</code>	Eight bytes (52-bit fractional mantissa, 11-bit exponent, sign bit)

19.2.5 Strings

Strings are encoded as a size (see Section 19.2.1), followed by the string contents in UTF8 format [21]. Strings are not NUL-terminated. An empty string is encoded with a size of zero.

19.2.6 Sequences

Sequences are encoded as a size (see Section 19.2.1) representing the number of elements in the sequence, followed by the elements encoded as specified for their type.

19.2.7 Dictionaries

Dictionaries are encoded as a size (see Section 19.2.1) representing the number of key–value pairs in the dictionary, followed by the pairs. Each key–value pair is

encoded as if it were a `struct` containing the key and value as members, in that order.

19.2.8 Enumerators

Enumerated values are encoded depending on the number of enumerators:

- If the enumeration has 1 - 127 enumerators, the value is marshaled as a `byte`.
- If the enumeration has 128 - 32767 members, the value is marshaled as a `short`.
- If the enumeration has more than 32767 members, the value is marshaled as an `int`.

The value is the ordinal value of the corresponding enumerator, with the first enumerator value encoded as zero.

19.2.9 Structures

The members of a structure are encoded in the order they appear in the `struct` declaration, as specified for their types.

19.2.10 Exceptions

Exceptions are marshaled as shown in Figure 19.1

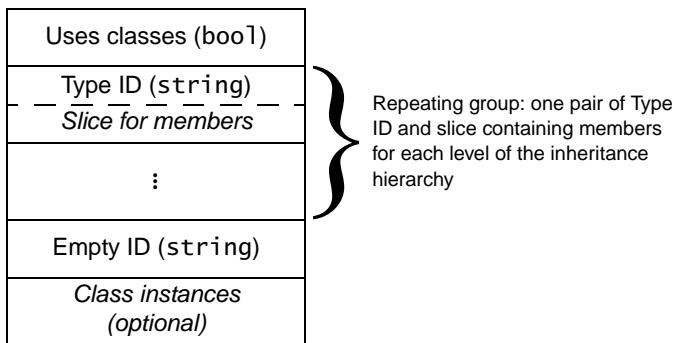


Figure 19.1. Marshaling format for exceptions.

Every exception instance is preceded by a single byte that indicates whether the exception uses class members: the byte value is 1 if any of the exception members

are classes (or if any of the exception members, recursively, contain class members) and 0, otherwise.

Following the header byte, the exception is marshaled as a sequence of pairs: the first member of each pair is the type ID for an exception slice, and the second member of the pair is a slice containing the marshaled members of that slice. The sequence of pairs is marshaled in derived-to-base order, with the most-derived slice first, and ending with the least-derived slice. The sequence of pairs is terminated by a single empty string. Within each slice, data members are marshaled as for structures: in the order in which they are defined in the Slice definition.

Following the sequence of pairs, any class instances that are used by the members of the exception are marshaled. This final part is optional: it is present only if the header byte is 1. (See Section 19.2.11 for a detailed explanation of how class instances are marshaled.)

To illustrate the marshaling, consider the following exception hierarchy:

```
exception Base {
    int baseInt;
    string baseString;
};

exception Derived extends Base {
    bool derivedBool;
    string derivedString;
    double derivedDouble;
};
```

Assume that the exception members are initialized to the values shown in Table 19.2.

Table 19.2. Member values of an exception of type `Derived`.

Member	Type	Value	Marshaled Size (in bytes)
<code>baseInt</code>	<code>int</code>	99	4
<code>baseString</code>	<code>string</code>	"Hello"	6
<code>derivedBool</code>	<code>bool</code>	true	1
<code>derivedString</code>	<code>string</code>	"World!"	7
<code>derivedDouble</code>	<code>double</code>	3.14	8

From Table 19.2, we can see that the total size of the members of `Base` is 10 bytes, and the total size of the members of `Derived` is 16 bytes. None of the exception members are classes. An instance of this exception has the on-the-wire representation shown in Table 19.3. (The size, type, and byte offset of the marshaled representation is indicated for each component.)

Table 19.3. Marshaled representation of the exception in Table 19.2.

Marshaled Value	Size in Bytes	Type	Byte offset
0 (<i>no class members</i>)	1	<code>bool</code>	0
" <code>: :Derived</code> " (<i>type ID</i>)	10	<code>string</code>	1
20 (<i>byte count for slice</i>)	4	<code>int</code>	11
1 (<i>derivedBool</i>)	1	<code>bool</code>	15
"World!" (<i>derivedString</i>)	7	<code>string</code>	16
3.14 (<i>derivedDouble</i>)	8	<code>double</code>	23
" <code>: :Base</code> " (<i>type ID</i>)	7	<code>string</code>	31
14 (<i>byte count for slice</i>)	4	<code>int</code>	38
99 (<i>baseInt</i>)	4	<code>int</code>	42
"Hello" (<i>baseString</i>)	6	<code>string</code>	46
" " (<i>empty string</i>)	1	<code>string</code>	52

Note that the size of each string is one larger than the actual string length. This is because each string is preceded by a count of its number of bytes, as explained in Section 19.2.5.

The receiver of this sequence of values uses the header byte to decide whether it eventually must unmarshal any class instances contained in the exception (none in this example) and then examines the first type ID (`: :Derived`). If the receiver recognizes that type ID, it can unmarshal the contents of the first slice, followed by the remaining slices; otherwise, the receiver reads the byte count that follows the unknown type (20) and then skips 20–4 bytes in the input stream, which is the start of the type ID for the second slice (`: :Base`). If the receiver does not recognize that type ID either, it again reads the byte count following the type ID (14), skips 14–4 bytes, and reads the empty type ID. This empty ID indicates the end of

the exception. If a receiver receives an exception without recognizing any of its type IDs, it throws an `UnknownUserException` to the application.

19.2.11 Classes

The marshaling for classes is complex, due to the need to deal with the pointer semantics for graphs of classes, as well as the need for the receiver to slice classes of unknown derived type. In addition, the marshaling for classes uses a type ID compression scheme to avoid repeatedly marshaling the same type IDs for large graphs of class instances.

Basic Marshaling Format

Classes are marshaled similar to exceptions: each instance is divided into a number of pairs containing a type ID and a slice (one pair for each level of the inheritance hierarchy) and marshaled in derived-to-base order. Only data members are marshaled—no information is sent that would relate to operations. Unlike exceptions, no header byte precedes a class. Instead, each marshaled class instance is preceded by a (non-zero) positive integer that provides an identity for the instance. The sender assigns this identity during marshaling such that each marshaled instance has a different identity. The receiver uses that identity to correctly reconstruct graphs of classes. Unlike for exceptions, the sequence of type ID and slice pairs is *not* terminated by an empty ID. The overall marshaling format for classes is shown in Figure 19.2.

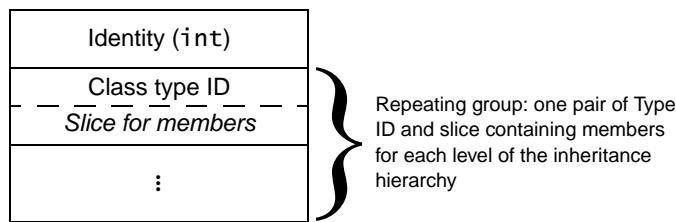


Figure 19.2. Marshaling format for classes.

Class Type IDs

Unlike for exception type IDs, class type IDs are not simple strings. Instead, a class type ID is marshaled as a boolean followed by either a string or a size, to conserve bandwidth. To illustrate this, consider the following class hierarchy:

```
class Base {  
    // ...  
};  
  
class Derived extends Base {  
    // ...  
};
```

The type IDs for the class slices are `::Derived` and `::Base`. Suppose the sender marshals three instances of `::Derived` as part of a single request. (For example, two instances could be out-parameters and one instance could be the return value.)

The first instance that is sent on the wire contains the type IDs `::Derived` and `::Base` preceding their respective slices. Because marshaling proceeds in derived-to-base order, the first type ID that is sent is `::Derived`. Every time the sender sends a type ID that it has not sent previously in the same request, it sends the boolean value `false`, followed by the type ID. Internally, the sender also assigns a unique positive number to each type ID. These numbers start at 1 and increment by one for each type ID that has not been marshaled previously. This means that the first type ID is encoded as the boolean value `false`, followed by `::Derived`, and the second type ID is encoded as the boolean value `false`, followed by `::Base`.

When the sender marshals the remaining two instances, it consults a lookup table of previously-marshaled type IDs. Because both type IDs were sent previously in the same request (or reply), the sender encodes all further occurrences of `::Derived` as the value `true` followed by the number 1 encoded as a size (see Section 19.2.1), and it encodes all further occurrences of `::Base` as the value `true` followed by the number 2 encoded as a size.

When the receiver reads a type ID, it first reads its boolean marker:

- If the boolean is `false`, the receiver reads a string and enters that string into a lookup table that maps integers to strings. The first new class type ID received in a request is numbered 1, the second new class type ID is numbered 2, and so on.
- If the boolean value is `true`, the receiver reads a number encoded as a size and uses that number to index into the lookup table to retrieve the corresponding class type ID.

Note that this numbering scheme is re-established for each new encapsulation. (As we will see in Section 19.3, parameters, return values, and exceptions are always marshaled inside an enclosing encapsulation.) For subsequent or nested encapsu-

lation, the numbering scheme restarts, with the first new type ID being assigned the value 1. In other words, each encapsulation uses its own independent numbering scheme for class type IDs to satisfy the constraint that encapsulations must not depend on their surrounding context.

Encoding class type IDs in this way provides significant savings in bandwidth: whenever an ID is marshaled a second and subsequent time, it is marshaled as a two-byte value (assuming no more than 254 distinct type IDs per request) instead of as a string. Because type IDs can be long, especially if you are using nested modules, the savings are considerable.

Simple Class Marshaling Example

To make the preceding discussion more concrete, consider the following class definitions:

```
interface SomeInterface {
    void op1();
};

class Base {
    int baseInt;
    void op2();
    string baseString;
};

class Derived extends Base implements SomeInterface {
    bool derivedBool;
    string derivedString;
    void op3();
    double derivedDouble;
};
```

Note that `Base` and `Derived` have operations, and that `Derived` also implements the interface `SomeInterface`. Because marshaling of classes is concerned with state, not behavior, the operations `op1`, `op2`, and `op3` are simply ignored during marshaling and the on-the-wire representation is as if the classes had been defined as follows:

```
class Base {
    int baseInt;
    string baseString;
};

class Derived extends Base {
```

```

    bool derivedBool;
    string derivedString;
    double derivedDouble;
};
```

Suppose the sender marshals two instances of `Derived` (for example, as two in-parameters in the same request). The member values are as shown in Table 19.4.

Table 19.4. Member values for two instances of class `Derived`.

	Member	Type	Value	_marshaled Size (in bytes)
First instance	baseInt	int	99	4
	baseString	string	"Hello"	6
	derivedBool	bool	true	1
	derivedString	string	"World! "	7
	derivedDouble	double	3.14	8
Second instance	baseInt	int	115	4
	baseString	string	"Cave"	5
	derivedBool	bool	false	1
	derivedString	string	"Canem"	6
	derivedDouble	double	6.32	8

The sender arbitrarily assigns a non-zero identity (see page 494) to each instance. Typically, the sender will simply consecutively number the instances starting at 1. For this example, assume that the two instances have the identities 1 and 2. The

marshaled representation for the two instances (assuming that they are marshaled immediately following each other) is shown in Table 19.5.

Table 19.5. Marshaled representation of the two instances in Table 19.4.

_marshaled Value	Size in Bytes	Type	Byte offset
1 (<i>identity</i>)	4	int	0
0 (<i>marker for class type ID</i>)	1	bool	4
"::Derived" (<i>class type ID</i>)	10	string	5
20 (<i>byte count for slice</i>)	4	int	15
1 (<i>derivedBool</i>)	1	bool	19
"World!" (<i>derivedString</i>)	7	string	20
3.14 (<i>derivedDouble</i>)	8	double	27
0 (<i>marker for class type ID</i>)	1	bool	35
"::Base" (<i>type ID</i>)	7	string	36
14 (<i>byte count for slice</i>)	4	int	43
99 (<i>baseInt</i>)	4	int	47
"Hello" (<i>baseString</i>)	6	string	51
0 (<i>marker for class type ID</i>)	1	bool	57
"::Ice::Object" (<i>class type ID</i>)	14	string	58
5 (<i>byte count for slice</i>)	4	int	72
0 (<i>number of dictionary entries</i>)	1	size	76
2 (<i>identity</i>)	4	int	77
1 (<i>marker for class type ID</i>)	1	bool	81
1 (<i>class type ID</i>)	1	size	82
19 (<i>byte count for slice</i>)	4	int	83
0 (<i>derivedBool</i>)	1	bool	87
"Canem" (<i>derivedString</i>)	6	string	88

Table 19.5. Marshaled representation of the two instances in Table 19.4.

Marshaled Value	Size in Bytes	Type	Byte offset
6 . 32 (<i>derivedDouble</i>)	8	<code>double</code>	94
1 (<i>marker for class type ID</i>)	1	<code>bool</code>	102
9 (<i>byte count for slice</i>)	4	<code>int</code>	103
2 (<i>class type ID</i>)	1	<code>size</code>	107
115 (<i>baseInt</i>)	4	<code>int</code>	108
"Cave" (<i>baseString</i>)	5	<code>string</code>	112
1 (<i>marker for class type ID</i>)	1	<code>bool</code>	117
3 (<i>class type ID</i>)	1	<code>size</code>	118
5 (<i>byte count for slice</i>)	4	<code>int</code>	119
0 (<i>number of dictionary entries</i>)	1	<code>size</code>	123

Note that, because classes (like exceptions) are sent as a sequence of slices, the receiver of a class can slice off any derived parts of a class it does not understand. Also note that (as shown in Table 19.5) each class instance contains three slices. The third slice is for the type `::Ice::Object`, which is the base type of all classes. The class type ID `::Ice::Object` has the number 3 in this example because it is the third distinct type ID that is marshaled by the sender. (See entries at byte offsets 58 and 118 in Table 19.5.) All class instances have this final slice of type `::Ice::Object`.

Marshaling a separate slice for `::Ice::Object` dates back to Ice versions 1.3 and earlier. In those versions, classes carried a facet map that was marshaled as if it were defined as follows:

```
module Ice {
    class Object;

    dictionary<string, Object> FacetMap;

    class Object {
        FacetMap facets; // No longer exists
    };
};
```

As of Ice version 1.4, this facet map is always empty, that is, the count of entries for the dictionary that is marshaled in the `::Ice::Object` slice is always zero. If a receiver receives a class instance with a non-empty facet map, it must throw a `MarshalException`.

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Marshaling Pointers

Classes support pointer semantics, that is, you can construct graphs of classes. It follows that classes can arbitrarily point at each other. The class identity (see page 494) is used to distinguish instances and pointers as follows:

- A class identity of 0 denotes a null pointer.
- A class identity > 0 precedes the marshaled contents of an instance (see page 494).
- A class identity < 0 denotes a pointer to an instance.

Identity values less than zero are pointers. For example, if the receiver receives the identity -57 , this means that the corresponding class member that is currently being unmarshaled will eventually point at the instance with identity 57 .

For structures, classes, exceptions, sequences, and dictionary members that do not contain class members, the Ice protocol uses a simple depth-first traversal algorithm to marshal the members. For example, structure members are marshaled in the order of their Slice definition; if a structure member itself is of complex type, such as a sequence, the sequence is marshaled in toto where it appears inside its enclosing structure. For complex types that contain class members, this depth-first marshaling is suspended: instead of marshaling the actual class instance at this point, a negative identity is marshaled that indicates which class instance that member must eventually denote. For example, consider the following definitions:

```
class C {
    // ...
};

struct S {
    int i;
    C firstC;
    C secondC;
    C thirdC;
    int j;
};
```

Suppose we initialize a structure of type S as follows:

```
S myS;
myS.i = 99;
myS.firstC = new C;           // New instance
myS.secondC = 0;             // null
myS.thirdC = myS.firstC;     // Same instance as previously
myS.j = 100;
```

When this structure is marshaled, the contents of the three class members are not marshaled in-line. Instead, the sender marshals the negative identities of the corresponding instances. Assuming that the sender has assigned the identity 78 to the instance assigned to myS.firstC, myS is marshaled as shown in Table 19.6.

Table 19.6. Marshaled representation of myS.

Marshaled Value	Size in Bytes	Type	Byte offset
99 (myS.i)	4	int	0
-78 (myS.firstC)	4	int	4
0 (myS.secondC)	4	int	8
-78 (myS.thirdC)	4	int	12
100 (myS.j)	4	int	16

Note that myS.firstC and myS.thirdC both use the identity -78. This allows the receiver to recognize that firstC and thirdC point at the same class instance (rather than at two different instances that happen to have the same contents).

Marshaling the negative identities instead of the contents of an instance allows the receiver to accurately reconstruct the class graph that was sent by the sender. However, this begs the question of *when* the actual instances are to be marshaled as described at the beginning of this section. As we will see in Section 19.3, parameters and return values are marshaled as if they were members of a structure. For example, if an operation invocation has five input parameters, the client marshals the five parameters end-to-end as if they were members of a single structure. If any of the five parameters are class instances, or are of complex type (recursively) containing class instances, the sender marshals the parameters in multiple passes: the first pass marshals the parameters end-to-end, using the usual depth-first algorithm:

- If the sender encounters a class member during marshaling, it checks whether it has marshaled the same instance previously for the current request or reply:
 - If the instance has not been marshaled before, the sender assigns a new identity to the instance and marshals the negative identity.
 - Otherwise, if the instance was marshaled previously, the sender sends the same negative identity that is previously sent for that instance.

In effect, during marshaling, the sender builds an identity table that is indexed by the address of each instance; the lookup value for the instance is its identity.

Once the first pass ends, the sender has marshaled all the parameters, but has not yet marshaled any of the class instances that may be pointed at by various parameters or members. The identity table at this point contains all those instances for which negative identities (pointers) were marshaled, so whatever is in the identity table at this point are the classes that the receiver still needs. The sender now marshals those instances in the identity table, but with positive identities and followed by their contents, as described on page 496. The outstanding instances are marshaled as a sequence, that is, the sender marshals the number of instances as a size (see Section 19.2.1), followed by the actual instances.

In turn, the instances just sent may themselves contain class members; when those class members are marshaled, the sender assigns an identity to new instances or uses a negative identity for previously marshaled instances as usual. This means that, by the end of the second pass, the identity table may have grown, necessitating a third pass. That third pass again marshals the outstanding class instances as a size followed by the actual instances. The third pass contains all those instances that were not marshaled in the second pass. Of course, the third pass may trigger yet more passes until, finally, the sender has sent all outstanding instances, that is, marshaling is complete. At this point, the sender terminates the sequence of passes by marshaling an empty sequence (the value 0 encoded as a size).

To illustrate this with an example, consider the definitions shown in Section 4.9.7 on page 104 once more:

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {
    idempotent long eval();
};
```

```

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand {
    long val;
};

```

These definitions allow us to construct expression trees. Suppose the client initializes a tree to the shape shown in Figure 19.3, representing the expression $(1 + 6 / 2) * (9 - 3)$. The values outside the nodes are the identities assigned by the client.

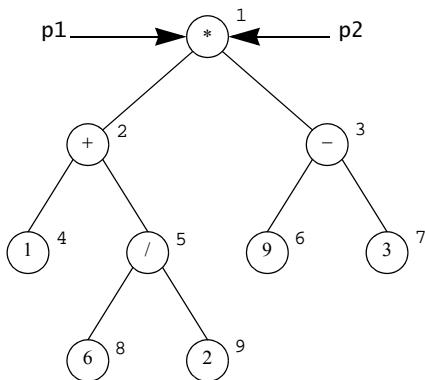


Figure 19.3. Expression tree for the expression $(1 + 6 / 2) * (9 - 3)$. Both `p1` and `p2` denote the root node.

The client passes the root of the tree to the following operation in the parameters `p1` and `p2`, as shown on page 503. (Even though it does not make sense to pass the same parameter value twice, we do it here for illustration purposes):

```

interface Tree {
    void sendTree(Node p1, Node p2);
};

```

The client now marshals the two parameters p1 and p2 to the server, resulting in the value -1 being sent twice in succession. (The client arbitrarily assigns an identity to each node. The value of the identity does not matter, as long as each node has a unique identity. For simplicity, the Ice implementation numbers instances with a counter that starts counting at 1 and increments by one for each unique instance.) This completes the marshaling of the parameters and results in a single instance with identity 1 in the identity table. The client now marshals a sequence containing a single element, node 1, as described on page 496. In turn, node 1 results in nodes 2 and 3 being added to the identity table, so the next sequence of nodes contains two elements, nodes 2 and 3. The next sequence of nodes contains nodes 4, 5, 6, and 7, followed by another sequence containing nodes 8 and 9. At this point, no more class instances are outstanding, and the client marshals an empty sequence to indicate to the receiver that the final sequence has been marshaled.

Within each sequence, the order in which class instances are marshaled is irrelevant. For example, the third sequence could equally contain nodes 7, 6, 4, and 5, in that order. What is important here is that each sequence contains nodes that are an equal number of “hops” away from the initial node: the first sequence contains the initial node(s), the second sequence contains all nodes that can be reached by traversing a single link from the initial node(s), the third sequence contains all nodes that can be reached by traversing two links from the initial node(s), and so on.

Now consider the same example once more, but with different parameter values for `sendTree`: `p1` denotes the root of the tree, and `p2` denotes the `-` operator of the right-hand sub-tree, as shown in Figure 19.4.

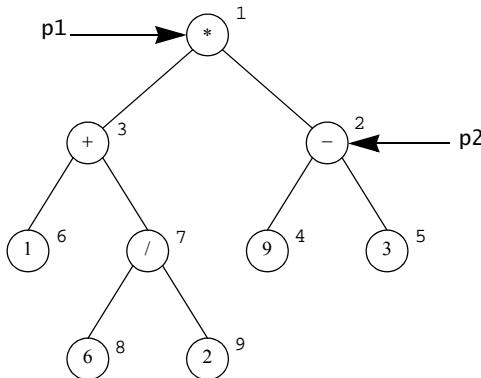


Figure 19.4. The expression tree of Figure 19.3, with `p1` and `p2` denoting different nodes.

The graph that is marshaled is exactly the same, but instances are marshaled in a different order and with different identities:

- During the first pass, the client sends the identities `-1` and `-2` for the parameter values.
- The second pass marshals a sequence containing nodes `1` and `2`.
- The third pass marshals a sequence containing nodes `3`, `4`, and `5`.
- The fourth pass marshals a sequence containing nodes `6` and `7`.
- The fifth pass marshals a sequence containing nodes `8` and `9`.
- The final pass marshals an empty sequence.

In this way, any graph of nodes can be transmitted (including graphs that contain cycles). The receiver reconstructs the graph by filling in a patch table during unmarshaling:

- Whenever the receiver unmarshals a negative identity, it adds that identity to a patch table; the lookup value is the memory address of the parameter or member that eventually will point at the corresponding instance.
- Whenever the receiver unmarshals an actual instance, it adds the instance to an unmarshaled table; the lookup value is the memory address of the instantiated class. The receiver then uses the address of the instance to patch any parameters or members with the actual memory address.

Note that the receiver may receive negative identities that denote class instances that have been unmarshaled already (that is, point “backward” in the unmarshaling stream), as well as instances that are yet to be unmarshaled (that is, point “forward” in the unmarshaling stream). Both scenarios are possible, depending on the order in which instances are marshaled, as well as their in-degree.

To provide another example, consider the following definition:

```
class C {  
    // ...  
};  
  
sequence<C> CSeq;
```

Suppose the client marshals a sequence of 100 C instances to the server, with each instance being distinct. (That is, the sequence contains 100 pointers to 100 different instances, not 100 pointers to the same single instance.) In that case, the sequence is marshaled as a size of 100, followed by 100 negative identities, -1 to -100. Following that, the client marshals a single sequence containing the 100 instances, each instance with its positive identity in the range 1 to 100, and completes by marshaling an empty sequence.

On the other hand, if the client sends a sequence of 100 elements that all point to the same single class instance, the client marshals the sequence as a size of 100, followed by 100 negative identities, all with the value -1. The client then marshals a sequence containing a single element, namely instance 1, and completes by marshaling an empty sequence.

Class Graphs and Slicing

It is important to note that when a graph of class instances is sent, it always forms a connected graph. However, when the receiver rebuilds the graph, it may end up with a disconnected graph, due to slicing. Consider:

```
class Base {  
    // ...  
};  
  
class Derived extends Base {  
    // ...  
    Base b;  
};
```

```
interface Example {  
    void op(Base p);  
};
```

Suppose the client has complete type knowledge, that is, understands both types `Base` and `Derived`, but the server only understands type `Base`, so the derived part of a `Derived` instance is sliced. The client can instantiate classes to be sent as parameter `p` as follows:

```
DerivedPtr p = new Derived;  
p->b = new Derived;  
ExamplePrx e = ...;  
e->op(p);
```

As far as the client is concerned, the graph looks like the one shown in Figure 19.5.

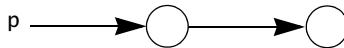


Figure 19.5. Sender-side view of a graph containing derived instances.

However, the server does not understand the derived part of the instances and slices them. Yet, the server unmarshals all the class instances, leading to the situation where the class graph has become disconnected, as shown in Figure 19.6.



Figure 19.6. Receiver-side view of the graph in Figure 19.5.

Of course, more complex situations are possible, such that the receiver ends up with multiple disconnected graphs, each containing many instances.

Exceptions with Class Members

If an exception contains class members, its header byte (see page 491) is 1 and the exception members are followed by the outstanding class instances as described on the preceding pages, that is, the actual exception members are followed by one or more sequences that contain the outstanding class instances, followed by an empty sequence that serves as an end marker.

19.2.12 Proxies

The first component of an encoded proxy is a value of type `Ice::Identity`. If the proxy is a nil value, the `category` and `name` members are empty strings, and no additional data is encoded. The encoding for a non-nil proxy consists of general parameters followed by endpoint parameters.

General Proxy Parameters

The general proxy parameters are encoded as if they were members of the following structure:

```
struct ProxyData {
    Ice::Identity id;
    Ice::StringSeq facet;
    byte mode;
    bool secure;
};
```

The general proxy parameters are described in Table 19.7.

Table 19.7. General proxy parameters.

Parameter	Description
<code>id</code>	The object identity
<code>facet</code>	The facet name (zero- or one-element sequence)
<code>mode</code>	The proxy mode (0=two-way, 1=one-way, 2=batch one-way, 3=datagram, 4=batch datagram)
<code>secure</code>	<code>true</code> if secure endpoints are required, otherwise <code>false</code>

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a proxy with a `facet` field with more than one element, it must throw a `ProxyUnmarshalException`.

Endpoint Parameters

A proxy optionally contains an endpoint list (see Appendix D) or an adapter identifier, but not both.

- If a proxy contains endpoints, they are encoded immediately following the general parameters. A size specifying the number of endpoints is encoded first (see Section 19.2.1), followed by the endpoints. Each endpoint is encoded as a short specifying the endpoint type (1=TCP, 2=SSL, 3=UDP), followed by an encapsulation (see Section 19.2.2) of type-specific parameters. The type-specific parameters for TCP, UDP, and SSL are presented in the sections that follow.
- If a proxy does not have endpoints, a single byte with value 0 immediately follows the general parameters and a string representing the object adapter identifier is encoded immediately following the zero byte.

Type-specific endpoint parameters are encapsulated because a receiver may not be capable of decoding them. For example, a receiver can only decode SSL endpoint parameters if it is configured with the SSL plug-in (see Chapter 24). However, the receiver must be able to re-encode the proxy with all of its original endpoints, in the order they were received, even if the receiver does not understand the type-specific parameters for an endpoint. Encapsulation of the parameters allows the receiver to do this.

TCP Endpoint Parameters

A TCP endpoint is encoded as an encapsulation containing the following structure:

```
struct TCPEndpointData {  
    string host;  
    int port;  
    int timeout;  
    bool compress;  
};
```

The endpoint parameters are described in Table 19.8.

Table 19.8. TCP endpoint parameters.

Parameter	Description
<code>host</code>	The server host (a host name or IP address)
<code>port</code>	The server port (1-65535)
<code>timeout</code>	The timeout in milliseconds for socket operations
<code>compress</code>	<code>true</code> if compression should be used (if possible), otherwise <code>false</code>

See Section 19.4 for more information on compression.

UDP Endpoint Parameters

A UDP endpoint is encoded as an encapsulation containing the following structure:

```
struct UDPEndpointData {
    string host;
    int port;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    bool compress;
};
```

The endpoint parameters are described in Table 19.9.

Table 19.9. UDP endpoint parameters.

Parameter	Description
<code>host</code>	The server host (a host name or IP address)
<code>port</code>	The server port (1-65535)
<code>protocolMajor</code>	The major protocol version supported by the endpoint
<code>protocolMinor</code>	The highest minor protocol version supported by the endpoint
<code>encodingMajor</code>	The major encoding version supported by the endpoint
<code>encodingMinor</code>	The highest minor encoding version supported by the endpoint
<code>compress</code>	<code>true</code> if compression should be used (if possible), otherwise <code>false</code>

See Section 19.4 for more information on compression.

SSL Endpoint Parameters

An SSL endpoint is encoded as an encapsulation containing the following structure:

```
struct SSLEndpointData {  
    string host;  
    int port;  
    int timeout;  
    bool compress;  
};
```

The endpoint parameters are described in Table 19.10.

Table 19.10. SSL endpoint parameters.

Parameter	Description
<code>host</code>	The server host (a host name or IP address)
<code>port</code>	The server port (1-65535)
<code>timeout</code>	The timeout in milliseconds for socket operations
<code>compress</code>	<code>true</code> if compression should be used (if possible), otherwise <code>false</code>

See Section 19.4 for more information on compression.

19.3 Protocol Messages

The Ice protocol uses five protocol messages:

- Request (from client to server)
- Batch request (from client to server)
- Reply (from server to client)
- Validate connection (from server to client)
- Close connection (client to server or server to client)

Of these messages, validate and close connection only apply to connection-oriented transports.

As with the data encoding described in Section 19.2, protocol messages have no alignment restrictions. Each message consists of a message header and (except for validate and close connection) a message body that immediately follows the header.

19.3.1 Message Header

Each protocol message has a 14-byte header that is encoded as if it were the following structure:

```
struct HeaderData {  
    int magic;  
    byte protocolMajor;  
    byte protocolMinor;  
    byte encodingMajor;  
    byte encodingMinor;  
    byte messageType;  
    byte compressionStatus;  
    int messageSize;  
};
```

The members are described in Table 19.11.

Table 19.11. Message header members.

Member	Description
<code>magic</code>	A four-byte magic number consisting of the ASCII-encoded values of ‘I’, ‘c’, ‘e’, ‘P’ (0x49, 0x63, 0x65, 0x50)
<code>protocolMajor</code>	The protocol major version number
<code>protocolMinor</code>	The protocol minor version number
<code>encodingMajor</code>	The encoding major version number
<code>encodingMinor</code>	The encoding minor version number
<code>messageType</code>	The message type
<code>compressionStatus</code>	The compression status of the message (see Section 19.4)
<code>messageSize</code>	The size of the message in bytes, including the header

Currently, both the protocol and the encoding are at version 1.0. The valid message types are shown in Table 19.12.

Table 19.12. Message types.

Message Type	Encoding
Request	0
Batch request	1
Reply	2
Validate connection	3
Close connection	4

The encoding for these message bodies of each of these message types is described in the sections that follow.

19.3.2 Request Message Body

A request message contains the data necessary to perform an invocation on an object, including the identity of the object, the operation name, and input parameters. A request message is encoded as if it were the following structure:

```
struct RequestData {
    int requestId;
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
    byte mode;
    Ice::Context context;
    Encapsulation params;
};
```

The members are described in Table 19.13.

Table 19.13. Request members.

Member	Description
<code>requestId</code>	The request identifier
<code>id</code>	The object identity
<code>facet</code>	The facet name (zero- or one-element sequence)
<code>operation</code>	The operation name
<code>mode</code>	A byte representation of <code>Ice::OperationMode</code> (0=normal, 1=nonmutating, 2=idempotent)
<code>context</code>	The invocation context
<code>params</code>	The encapsulated input parameters, in order of declaration

The request identifier zero (0) is reserved for use in oneway requests and indicates that the server must not send a reply to the client. A non-zero request identifier must uniquely identify the request on a connection, and must not be reused while a reply for the identifier is outstanding.

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a request with a `facet` field with more than one element, it must throw a `MarshalException`.

19.3.3 Batch Request Message Body

A batch request message contains one or more oneway requests, bundled together for the sake of efficiency. A batch request message is encoded as if it were a sequence of the following structure:

```
struct BatchrequestData {
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
    byte mode;
    Ice::Context context;
    Encapsulation params;
};
```

The members are described in Table 19.14.

Table 19.14. Batch request members.

Member	Description
<code>id</code>	The object identity
<code>facet</code>	The facet name (zero- or one-element sequence)
<code>operation</code>	The operation name
<code>mode</code>	A byte representation of <code>Ice::OperationMode</code>
<code>context</code>	The invocation context
<code>params</code>	The encapsulated input parameters, in order of declaration

Note that no request ID is necessary for batch requests because only oneway invocations can be batched.

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a batch request with a `facet` field with more than one element, it must throw a `MarshalException`.

19.3.4 Reply Message Body

A reply message body contains the results of a twoway invocation, including any return value, `out`-parameters, or exception. A reply message body is encoded as if it were the following structure:

```
struct ReplyData {
    int requestId;
    byte replyStatus;
    // [... messageSize - 19 bytes ...]
};
```

The first four bytes of a reply message body contain a request ID. The request ID matches an outgoing request and allows the requester to associate the reply with the original request (see Section 19.3.2).

The byte following the request ID indicates the status of the request; the remainder of the reply message body following the status byte depends on the status value. The possible status values are shown in Table 19.15.

Table 19.15. Reply status.

Reply status	Encoding
Success	0
User exception	1
Object does not exist	2
Facet does not exist	3
Operation does not exist	4
Unknown Ice local exception	5
Unknown Ice user exception	6
Unknown exception	7

Reply Status 0: Success

A successful reply message is encoded as an encapsulation containing `out`-parameters (in the order of declaration), followed by the return value for the invocation, encoded according to their types as specified in Section 19.2. If an operation declares a `void` return type and no `out`-parameters, an empty encapsulation is encoded.

Reply Status 1: User exception

A user exception reply message contains an encapsulation containing the user exception, encoded as described in Section 19.2.10.

Reply Status 2: Object does not exist

If the target object does not exist, the reply message is encoded as if it were the following structure:

```
struct ReplyData {
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
};
```

The members are described in Table 19.16.

Table 19.16. Invalid object reply members.

Member	Description
<code>id</code>	The object identity
<code>facet</code>	The facet name (zero- or one-element sequence)
<code>operation</code>	The operation name

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a reply with a `facet` field with more than one element, it must throw a `MarshalException`.

Reply Status 3: Facet does not exist

If the target object does not support the facet encoded in the request message, the reply message is encoded as for reply status 2.

Reply Status 4: Operation does not exist

If the target object does not support the operation encoded in the request message, the reply message is encoded as for reply status 2.

Reply Status 5: Unknown Ice local exception

The reply message for an unknown Ice local exception is encoded as a single string that describes the exception.

Reply Status 6: Unknown Ice user exception

The reply message for an unknown Ice user exception is encoded as a single string that describes the exception.

Reply Status 7: Unknown exception

The reply message for an unknown exception is encoded as a single string that describes the exception.

19.3.5 Validate Connection Message

A server sends a validate connection message when it receives a new connection.¹ The message indicates that the server is ready to receive requests; the client must not send any messages on the connection until it has received the validate connection message from the server. No reply to the message is expected by the server.

The purpose of the validate connection message is two-fold:

- It informs the client of the protocol and encoding versions that are supported by the server (see Section 19.5.3).
- It prevents the client from writing a request message to its local transport buffers until after the server has acknowledged that it can actually process the request. This avoids a race condition caused by the server’s TCP/IP stack accepting connections in its backlog while the server is in the process of shutting down: if the client were to send a request in this situation, the request would be lost but the client could not safely re-issue the request because that might violate at-most-once semantics.

The validate connection message guarantees that a server is not in the middle of shutting down when the server’s TCP/IP stack accepts an incoming connection and so avoids the race condition.

The message header described in Section 19.3.1 on page 512 comprises the entire validate connection message. The compression status of a validate connection message is always 0.

19.3.6 Close Connection Message

A close connection message is sent when a peer is about to gracefully shutdown a connection.² The message header described in Section 19.3.1 comprises the entire close connection message. The compression status of a close connection message is always 0.

1. Validate connection messages are only used for connection-oriented transports.

2. Close connection messages are only used for connection-oriented transports.

Either client or server can initiate connection closure. On the client side, connection closure is triggered by *Active Connection Management (ACM)* (see XREF), which automatically reclaims connections that have been idle for some time.

This means that connection closure can be initiated at will by either end of a connection; most importantly, no state is associated with a connection as far as the object model or application semantics are concerned.

The client side can close a connection whenever no reply for a request is outstanding on the connection. The sequence of events is:

1. The client sends a close connection message.
2. The client closes the writing end of the connection.
3. The server responds to the client's close connection message by closing the connection.

The server side can close a connection whenever no operation invocation is in progress that was invoked via that connection. This guarantees that the server will not violate at-most-once semantics: an operation, once invoked in a servant, is allowed to complete and its results are returned to the client. Note that the server can close a connection even after it has received a request from the client, provided that the request has not yet been passed to a servant. In other words, if the server decides that it wants to close a connection, the sequence of events is:

1. The server discards all incoming requests on the connection.
2. The server waits until all still executing requests have completed and their results have been returned to the client.
3. The server sends a close connection message to the client.
4. The server closes its writing end of the connection.
5. The client responds to the server's close connection message by closing both its reading and writing ends of the connection.
6. If the client has outstanding requests at the time it receives the close connection message, it re-issues these requests on a new connection. Doing so is guaranteed not to violate at-most-once semantics because the server guarantees not to close a connection while requests are still in progress on the server side.

19.3.7 Protocol State Machine

From a client's perspective, the Ice protocol behaves according to the state machine shown in Figure 19.7.

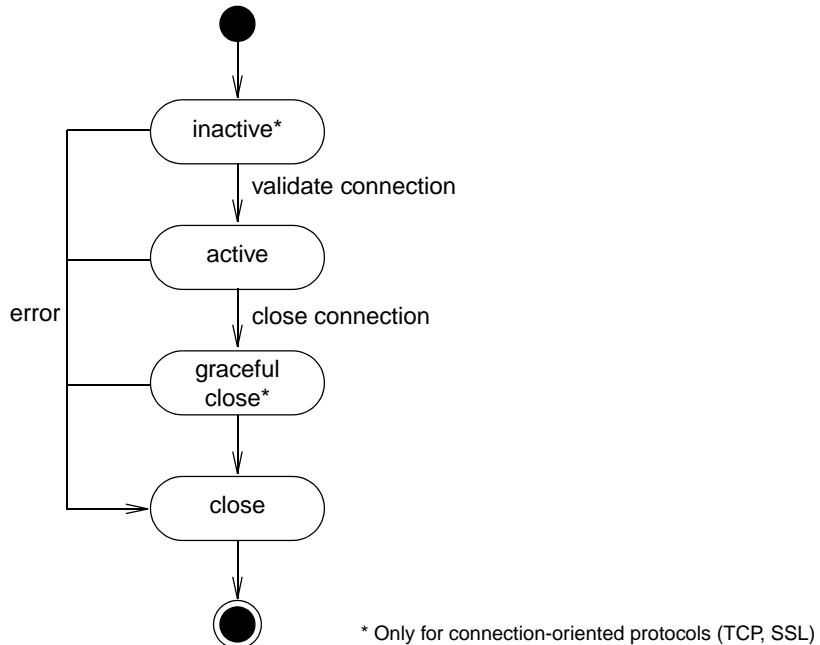


Figure 19.7. Protocol state machine.

To summarize, a new connection is inactive until a validate connection message (see Section 19.3.5) has been received by the client, at which point the active state is entered. The connection remains in the active state until it is shut down, which can occur when there are no more proxies using the connection, or after the connection has been idle for a while. At this point, the connection is gracefully closed, meaning that a close connection message is sent (see Section 19.3.6), and the connection is closed.

19.3.8 Disorderly Connection Closure

Any violation of the protocol or encoding rules results in a disorderly connection closure: the side of the connection that detects a violation unceremoniously closes it (without sending a close connection message or similar). There are many poten-

tial error conditions that can lead to disorderly connection closure; for example, the receiver might detect that a message has a bad magic number or incompatible version, receive a reply with an ID that does not match that of an outstanding request, receive a validate connection message when it should not, or find illegal data in a request (such as a negative size, or a size that disagrees with the actual data that was unmarshaled).

19.4 Compression

Compression is an optional feature of the Ice protocol; whether it is used for a particular message is determined by several factors:

1. Compression may not be supported on all platforms or in all language mappings. For example, compression is currently not supported by Ice for Java.
2. Compression can be used in a request or batch request only if the proxy specifies it (see Section 19.2.12).
3. A reply message is compressed only if its request was compressed.
4. For efficiency reasons, the Ice protocol engine does not compress messages smaller than 100 bytes.

If compression is used, the entire message excluding the header is compressed using the bzip2 algorithm [16]. The `messageSize` member of the message header therefore reflects the size of the compressed message, including the uncompressed header.

The `compressionStatus` field of the message header (see Section 19.3.1) indicates whether a message is compressed and whether the sender can accept a compressed reply, as shown in Table 19.17.

Table 19.17. Compression status values.

Reply status	Encoding	Applies to
Message is uncompressed, sender cannot accept a compressed reply.	0	Request, Batch Request, Reply, Validate Connection, Close Connection
Message is uncompressed, sender can accept a compressed reply.	1	Request, Batch Request
Message is compressed and sender can accept a compressed reply	2	Request, Batch Request, Reply

- A compression status of 0 indicates that the message is not compressed and, moreover, that the sender of this message cannot accept a compressed reply. A client that does not support compression always uses this value. A client that supports compression sets the value to 0 if the endpoint via which the request is dispatched indicates that it does not support compression.

A server uses this value for uncompressed replies.

- A compression status of 1 indicates that the message is not compressed, but that the server is free to return a compressed reply. A client uses this value if the endpoint via which the request is dispatched indicates that it supports compression, but the client has decided not to use compression for this particular request (presumably because the request is too small, so compression does not provide any saving).

This value applies only to request and batch request messages.

- A compression status of 2 indicates that the message is compressed and that the server is free to reply with a compressed message (but need not reply with a compressed message). A client that supports compression (obviously) sets this value only if the endpoint via which the request is dispatched indicates that it supports compression.

A server uses this value for compressed replies.

The message body of compressed request, batch request, or reply message is consists of the compressed version of the encodings specified in Sections 19.3.2 through 19.3.4.

Note that compression is likely to improve performance only over lower-speed links, for which bandwidth is the overall limiting factor. Over high-speed LAN links, the CPU time spent on compressing and uncompressing messages is longer than the time it takes to just send the uncompressed data.

19.5 Protocol and Encoding Versions

As we saw in the preceding sections, both the Ice protocol and encoding have separate major and minor version numbers. Separate versioning of protocol and encoding has the advantage that neither depends on the other: any version of the Ice protocol can be used with any version of the encoding, so they can evolve independently. (For example, Ice protocol version 1.1 could use encoding version 2.3, and vice versa.)

The Ice versioning mechanism provides the maximum possible amount of interoperability between clients and servers that use different versions of the Ice run time. In particular, older deployed clients can communicate with newer deployed servers and vice versa, provided that the message contents use types that are understandable to both sides.

For an example, assume that a later version of Ice were to introduce a new Slice keyword and data type, such as `complex`, for complex numbers. This would require a new minor version number for the encoding; let us assume that version 1.1 of the encoding is identical to the 1.0 encoding but, in addition,

supports the `complex` type. We now have four possible combinations of client and server encoding versions:

Table 19.18. Interoperability for different versions.

Client Version	Server Version	Operation with complex Parameter	Operation without complex Parameter
1.0	1.0	N/A	✓
1.1	1.0	N/A	✓
1.0	1.1	N/A	✓
1.1	1.1	✓	✓

As you can see, interoperability is provided to the maximum extent possible. If both client and server are at version 1.1, they can obviously exchange messages and will use encoding version 1.1. For version 1.0 clients and servers, obviously only operations that do not involve `complex` parameters can be invoked (because at least one of client and server do not know about the new `complex` type) and messages are exchanged using encoding version 1.0.

19.5.1 Version Ground Rules

For versioning of the protocol and encoding to be possible, *all* versions (present and future) of the Ice run time adhere to a few ground rules:

1. Encapsulations always have a six-byte header; the first four bytes are the size of the encapsulation (including the size of the header), followed by two bytes that indicate the major and minor version. How to interpret the remainder of the encapsulation depends on the major and minor version.
2. The first eight bytes of a message header always contain the magic number 'I', 'c', 'e', 'P', followed by four bytes of version information (two bytes for the protocol major and minor number, and two bytes of the encoding major and minor number). How to interpret the remainder of the header and the message body depends on the major and minor version.

These ground rules ensure that all current and future versions of the Ice run time can at least identify the version and size of an encapsulation and a message. This is particularly important for message switches such as IceStorm (see Chapter 27);

by keeping the version and size information in a fixed format, it is possible to forward messages that are, for example, at version 2.0, even though the message switch itself may still be at version 1.0.

19.5.2 Version Compatibility Rules

To establish whether a particular protocol version is compatible with another protocol version (or a particular encoding version is compatible with another encoding version), the following rules apply:

1. Different major versions are incompatible. There is no obligation on either clients or servers to support more than a single major version. For example, a server with major version 2 is under no obligation to also support major version 1.

This rule exists to permit the Ice run time to eventually get rid of old versions—without such a rule, all future releases of Ice would have to support all previous major versions forever. In plain language, the rule means that clients and servers that use different major versions simply cannot communicate with each other.

2. A receiver that advertises minor version n guarantees to be able to successfully decode all minor versions less than n . Note that this does *not* imply that messages using version $n-1$ can be decoded as if they were version n : as far as their physical representation is concerned, two adjacent minor versions can be completely incompatible. However, because any receiver advertising version n is also obliged to correctly deal with version $n-1$, minor version upgrades are *semantically* backward compatible, even though their physical representation may be incompatible.
3. A sender that supports minor version n guarantees to be able to send messages using all minor versions less than n . Moreover, the sender guarantees that if it receives a request using minor version k (with $k \leq n$), it will send the reply for that request using minor version k .

19.5.3 Version Negotiation

Client and server must somehow agree on which version to use to exchange messages. Depending on whether the underlying transport is connection-oriented or connection-less, different mechanisms are used to negotiate a common version.

Negotiation for Connection-Oriented Transports

For connection-oriented transports, the client opens a connection to the server and then waits for a validate connection message (see page 519). The validate connection message sent by the server indicates the server's major and highest supported minor version numbers for both protocol and encoding. If the server's and client's major version numbers do not match, the client side raises an `UnsupportedProtocolException` or `UnsupportedEncodingException`.

Assuming that the client has received a validate connection message from the server that matches the client's major version, the client knows the highest minor version number that is supported by the server. Thereafter, the client is obliged to send no message with a minor version number higher than the server's limit. However, the client is free to send a message with a minor version number that is less than the server's limit.

The server does not have a-priori knowledge of the highest minor version that is supported by the client (because there is no validate connection message from client to server). Instead, the server learns about the client version number in each individual message, by looking at the message header. That minor version indicates the minor version number that the client can accept. The scope of that minor version number is a single request-reply interaction. For example, if the client sends a request with minor version 3, the server must reply to that request with minor version 3 as well. However, the next client request might be with minor version 2, and the server must reply to that request with minor version 2.

For orderly connection closure via a close connection message, the server can use any minor version, but that minor version must not be higher than the highest minor version number that was received from the client while the connection was open.

Negotiation for Connection-Less Transports

For connection-less transports, no validate connection message exists, so the client must learn about the highest supported minor version number of the server via other means. The mechanism for this depends on whether a proxy for a connection-less endpoint is bound directly or indirectly (see Section 21.3.2):

- For direct proxies, the version information is part of the endpoint contained in the proxy. In this case, the client simply sends its messages with a minor version number that is not greater than the minor version number of the endpoint in the proxy.
- For indirect proxies, the proxy itself contains no version information at all (because the proxy contains no endpoints). Instead, the client obtains the

version information when it resolves the adapter name to one or more endpoints (via IcePack or an equivalent service). The version information of the endpoints determines the highest minor version number that is available to the client.

19.6 A Comparison with IIOP

It is interesting to compare the Ice protocol and encoding with CORBA's Inter-ORB Interoperability Protocol (IIOP) and Common Data Representation (CDR) encoding. The Ice protocol and encoding differ from IIOP and CDR in a number of important respects:

- Fewer data types

CORBA IDL has data types that are not provided by Ice, such as characters and wide characters, fixed-point numbers, arrays, and unions, each of which require a separate set of encoding rules.

Obviously, fewer data types makes the Ice encoding more efficient and less complex than CDR.

- Fixed byte-order marshaling

CDR supports both big-endian and little-endian encoding and provides a “receiver-makes-it-right” approach to byte ordering. The advantage of this is that, if sender and receiver have matching byte order, no reordering of the data is required at either end. However, the disadvantage is the additional complexity this creates in the marshaling logic. Moreover, the “receiver-makes-it-right” scheme carries a severe performance penalty if messages are to be forwarded via a number of intermediaries because this can require repeated unmarshaling, reordering, and remarshaling of messages. (Encapsulating the data would avoid this problem but, unfortunately, IIOP does not encapsulate most of the data that would benefit from it.)

The Ice encoding uses fixed little-endian data layout and avoids both the complexity and performance penalty.

- No padding

CDR has a complex set of rules for inserting alignment bytes into a data stream in an attempt to keep data aligned in a way that suits the native data layout of the underlying hardware. Unfortunately, this approach is severely flawed for a number of reasons:

- The CDR padding rules actually do not achieve any layout on natural word boundaries. For example, depending on its relative position in the enclosing data stream, the same structure value can differ in size and padding by up to seven bytes. There is not a single hardware platform in existence whose layout rules are the same as those of CDR. As a result, the padding bytes that are inserted serve no purpose other than to waste bandwidth.
- With “receiver-makes-it-right”, the receiver is obliged to re-order data if it arrives in the incorrect byte order anyway. This means that any alignment of the data (even if it were correct) is useless to, on average, half of all receivers because re-ordering of the data requires copying all of the data anyway.
- Padding and alignment rules differ depending on the hardware platform as well as the compiler. (For example, many compilers provide options that change the way data is packed in memory, to allow the developer to control the time vs. space trade-off.) This means that even the best set of layout rules can suit only a minority of platforms.
- Data alignment rules greatly complicate data forwarding. For example, if the receiver of a data item wants to forward that data item to another downstream receiver, it cannot just block-copy the received data into its transmit buffer because the padding of the data is sensitive to its relative position in the enclosing byte stream and a block copy would most likely result in an illegal encoding.

Ice avoids all the complexity (and concomitant inefficiency) by aligning all data on byte boundaries.

- More compact encoding

CDR encoding rules are wasteful of bandwidth, especially for sequences of short data items. For example, with CDR encoding, an empty string occupies eight bytes: four bytes to store the string length, plus a single terminating NUL byte, plus three bytes of padding. Table 19.19 compares the encoding sizes of

CDR and the Ice encoding for sequences of 100 strings for different string lengths.

Table 19.19. Sizes of CDR and Ice sequences of 100 strings of different lengths.

Sequence of 100 Strings of Length	CDR Size	Ice Size	Extra cost of CDR
0	804	101	696%
4	1204	501	140%
8	1604	901	78%
16	2004	1701	18%

Similar savings are achieved for small structures. Depending on the order and type of the structure members, CDR padding bytes can almost double the structure size, which becomes significant when sending sequences of such structures.

- Simple proxy encoding

Due to the inability of CORBA vendors to agree on a common encoding for object references (the equivalent of Ice proxies), CORBA object references have a complex internal structure that is partially standardized and partially opaque, allowing vendors to add proprietary information to object references. In addition, to avoid object references getting too large, references that support more than one transport have a scheme for sharing the object identity among several transports instead of carrying multiple copies of the same identity. The encoding that is required to support all this machinery is quite complex and results in extremely poor marshaling performance when large number of object references are exchanged (for example, when using a trading service).

In contrast, Ice proxies are simple and straight-forward to marshal and do not incur this loss of performance.

- Proper version negotiation

Versioning for IIOP and CDR was never designed properly, with the result that version negotiation in IIOP simply does not work. In particular, CDR encapsulations do not carry a separate version number. As a result, it is possible for data in encapsulations to travel to receivers that cannot decode the contents of the encapsulation, with no mechanism at the protocol level to detect the problem. Lack of correct versioning has been an ongoing problem with CORBA for years and the problem has historically been dealt with by pretending that it does not exist (meaning that different CORBA versions cannot interoperate with each other in many circumstances).

The Ice protocol and encoding have well-defined versioning rules that avoid such problems, allow both protocol and encoding to be extended, and reliably detect version mismatches.

- Fewer message types

IIOP has more message types than Ice. For example, IIOP has both a cancel request and a message error message. The cancel request is meant to cancel an invocation in progress but, of course, cannot do that because there is no way to abort an executing invocation in the server. At best, a cancel request allows the server to avoid marshaling the results of an invocation back to the client. However, the additional complexity introduced into the protocol is not worth the saving. (And, at any rate, neither is there a way for an application developer to send a cancel request, nor can the run time decide on its own when it would be appropriate to send one; despite that, every compliant CORBA implementation is burdened by the need to correctly respond to a useless request.)

IIOP's message error message constitutes similar baggage: the receiver of a malformed message is obliged to respond with a message error message before closing its connection. However, the message carries no useful information and, due to the nature of TCP/IP implementations, is usually lost instead of being delivered. This means that a compliant CORBA implementation is forced to send a useless message that will not be received in most cases when, instead, it should simply close the connection.

Ice avoids any such baggage in its protocol: the messages that are used actually serve a useful purpose.

- Request batching

IIOP has no notion of request batching. The advantages of request batching are particularly noticeable for event forwarding mechanisms, such as IceS-

torm (see Chapter 27), as well as for fine-grained interfaces that provide modifier operations for a number of attributes. Request batching significantly reduces networking overhead for such applications.

- Reliable connection establishment

IIOP is vulnerable to connection loss during connection establishment. In particular, when a client opens a connection to a server, the client has no way of knowing whether the server is about to shut down and will not be able to process an incoming request. This means that the client has no choice but to send a request on a newly-established connection in the hope that the request will actually be processed by the server. If the server can process the request, there is no problem. However, if the server cannot process the request because it is on its way down, the client is confronted with a broken connection and cannot re-issue the request because that might violate at-most-once semantics.

Ice does not have this problem because the validate connection message ensures that the client will not send a request to a server that is about to shut down. Moreover, any requests with outstanding replies can safely be re-issued by the client without violating at-most-once semantics.

- Reliable endpoint resolution

Indirect binding (see Section 21.3.2) in IIOP relies on a locate forward reply. Briefly, endpoint resolution is transparent to clients using IIOP. If an object reference uses indirect binding, the client issues the request as usual and receives a locate forward reply containing the endpoint of the actual server. In response to that reply, the client issues the request a second time to contact the actual server. There are a number of problems with this scheme:

- The physical address of the location service is written into each indirectly bound object reference. This makes it impossible to move the location service without invalidating all the references held by clients.³

Ice does not have this problem because the location service is known to clients through configuration. If the location service is moved, clients can be updated by changing their configuration and there is no need to track down a potentially unlimited number of proxies that might be out of date. Moreover,

3. IIOP version 1.2 supported a message to indicate permanent location forwarding. That message was meant to ease migration of location services. However, the semantics of that message broke the object model elsewhere, with the result that IIOP version 1.3 has deprecated that message again. (Unfortunately, reversals such as this are all too common in OMG specifications.)

there is typically no need to move the location service. Instead, it is possible to construct federated location services that work similar to the Internet Domain Name Service (DNS) and internally forward requests to the correct resolver.

- In order to get a location forward message, clients send the actual request to the location service. This is expensive if the request parameters are large because they are marshaled twice: once to the location service and once to the actual server. IIOP adds a locate request message that allows a client to explicitly resolve the location of a server. However, CORBA object references carry no indication as to whether they are bound directly or indirectly. This means that, no matter what the client does, it is wrong some of the time: if the client always uses a locate request with a directly bound reference, it ends up incurring the cost of two remote messages instead of one; if the client sends the request directly with an indirectly bound reference, it incurs the cost of marshaling the parameters twice instead of once.

Ice makes location resolution a step that is explicitly visible to the client-side run time because proxies either carry endpoint information (for direct proxies) or an adapter name (for indirect proxies). This allows the client-side run time to select the correct resolution mechanism without having to play guessing games and to avoid the overhead incurred by location forwarding.

- With IIOP, location resolution is built into the protocol itself. This complicates the protocol with two additional message types and (until 2002, when additional APIs were added to CORBA) made it impossible to implement a location service using standard APIs.

With Ice, no special protocol support is required for location resolution. Instead, the location service is an ordinary Ice server that defines an interface as usual and is contacted using operation invocations like any other server.

- No codeset negotiation

IIOP uses a codeset negotiation feature that (supposedly) permits use of arbitrary character encodings for transmission of wide characters and strings, provided sender and receiver have at least one codeset in common. Unfortunately, this feature was never properly thought through and has repeatedly led to interoperability problems. (Every single version of the CORBA specification, including the latest 3.0 version, has made corrections to the way codeset negotiation is supposed to work. Whether the latest set of corrections will

succeed in dealing with the problem remains to be seen. Regardless, many pages of complexity (and many lines of ORB source code) are devoted to this (mis)feature.)

The UTF-8 encoded Unicode used by Ice avoids all these problems without losing any functionality.

- No fragmentation

IIOP provides a fragment message whose purpose it is to reduce memory overhead in the server during marshaling of the results of an operation invocation. Unfortunately, the feature is quite complex (having been mis-specified and mis-implemented repeatedly in the past). The savings to be gained through fragmentation are quite limited, yet every client-side ORB implementation is forced to provide support for the feature. (In other words, the cure is worse than the disease.)

Ice simply does not use a fragmentation scheme, avoiding both the complexity and the resulting code bloat.

- Support for connection-less transports

In 2001, the OMG adopted a multi-cast specification. To the best of our knowledge, as of early 2003, no implementations of that specification are available.

Ice offers UDP as an alternative to TCP/IP. For messaging services, such as IceStorm, the performance benefits of UDP are considerable, allowing the service to scale well beyond what could be achieved with TCP/IP.

Chapter 20

Ice Extension for PHP

20.1 Chapter Overview

This chapter describes IcePHP, the Ice extension for the PHP scripting language. Section 20.2 provides an overview of IcePHP, including its design goals, capabilities, and limitations. IcePHP configuration is discussed in Section 20.3, and the PHP language mapping is specified in Section 20.4.

20.2 Introduction

PHP is a general-purpose scripting language that is used primarily in Web development. The PHP interpreter is typically installed as a Web server plug-in, and PHP itself also supports plug-ins known as “extensions.” PHP extensions, by definition, extend the interpreter’s run-time environment by adding new functions and data types.

The Ice extension for PHP, *IcePHP*, provides PHP scripts with access to Ice facilities. IcePHP is a thin integration layer implemented in C++ using the Ice C++ run-time library. This implementation technique has a number of advantages over a native PHP implementation of the Ice run-time:

1. Speed

The majority of the time-consuming work involved in making remote invocations, such as marshaling and unmarshaling, is performed in compiled C++, instead of in interpreted PHP.

2. Integration

IcePHP is fully self-contained. Its installation is performed once as an administrative step, and scripts have no dependencies on external PHP code.

3. Reliability

By leveraging the well-tested Ice C++ run-time library, there is less likelihood that new bugs will be introduced into the PHP extension.

4. Flexibility

IcePHP inherits all of the flexibility provided by the Ice C++ run-time, such as support for SSL, protocol compression, etc.

20.2.1 Capabilities

IcePHP supplies a robust subset of the Ice run-time facilities. PHP scripts are able to use all of the Slice data types in a natural way (see Section 20.4), make remote invocations, and use all of the advanced Ice services such as routers, locators and protocol plug-ins.

20.2.2 Limitations

The primary design goal of IcePHP was to provide PHP scripts with a simple and efficient interface to the Ice run-time. To that end, the feature set supported by IcePHP was carefully selected to address the requirements of typical PHP applications. As a result, IcePHP does not support the following Ice features:

- Servers

Given PHP's primary role as a scripting language for dynamic Web pages, the ability to implement an Ice server in PHP was deemed unnecessary for the majority of PHP applications.

- Asynchronous method invocation

The lack of synchronization primitives in PHP greatly reduces the utility of asynchronous invocations.

- Multiple communicators

A script has access to only one instance of `Ice::Communicator`, and is not able to manually create or destroy a communicator. See Section 20.4.11 for more information.

20.2.3 Design

The traditional design for a language mapping requires the intermediate step of translating Slice definitions into the target programming language before they can be used in an application.

IcePHP takes a different approach, made possible by the flexibility of PHP's extension interfaces. In IcePHP, no intermediate code generation step is necessary. Instead, the extension is configured with the application's Slice definitions, which are used to drive the extension's run-time behavior (see Section 20.3).

The Slice definitions are made available to PHP scripts as specified by the mapping in Section 20.4, just as if the Slice definitions had first gone through the traditional code-generation step and then been imported by the script.

There are several advantages to this design:

- The development process is simplified by the elimination of the intermediate code-generation step.
- The lack of machine-generated PHP code reduces the risk that the application's type definitions become outdated.
- Although PHP is a loosely-typed programming language, the Slice definitions enable IcePHP to validate the arguments of remote invocations.

20.3 Configuration

This section defines the PHP configuration directives supported by IcePHP. For installation instructions, please refer to the `INSTALL` file included in the IcePHP distribution.

20.3.1 Profiles

IcePHP allows any number of PHP applications to run independently in the same PHP interpreter, without risk of conflicts caused by Slice definitions that happened to use the same identifiers. IcePHP uses the term *profile* to describe an

application's configuration, including its Slice definitions and Ice configuration properties.

20.3.2 Default Profile

A default profile is supported, which is convenient during development, or when only one Ice application is running in a PHP interpreter. Table 20.1 describes the PHP configuration directives¹ for the default profile.

Table 20.1. PHP configuration directive for the default profile.

Name	Description
ice.config	Specifies the pathname of an Ice configuration file.
ice.options	Specifies command-line options for Ice configuration properties. For example, --Ice.Trace.Network=1. This is a convenient alternative to an Ice configuration file if only a few configuration properties are required.
ice.profiles	Specifies the pathname of a profile configuration file. See Section 20.3.3.
ice.slice	Specifies preprocessor options and the pathnames of Slice files to be loaded.

Here is a simple example:

```
ice.options="--Ice.Trace.Network=1 --Ice.Warn.Connections=1"
ice.slice="-I/myapp/include /myapp/include/MyApp.ice"
```

20.3.3 Named Profiles

If a filename is specified by the `ice.profiles` configuration directive, the file is expected to have the standard INI-file format. The section names identify

1. These directives are typically defined in the `php.ini` file, but can also be defined using Web-server specific directives.

profiles, where each profile supports the `ice.config`, `ice.options` and `ice.slice` directives defined in Section 20.3.2. For example:

```
[Profile1]
ice.config=/profile1/ice.cfg
ice.slice=/profile1/App.ice
```

```
[Profile2]
ice.config=/profile2/ice.cfg
ice.slice=/profile2/App.ice
```

This file defines two named profiles, `Profile1` and `Profile2`, having separate Ice configurations and Slice definitions.

20.3.4 Profile Functions

Two global functions are provided for profile activities:

```
Ice_loadProfile(/* string */ $name = null);
Ice_dumpProfile();
```

The `Ice_loadProfile` function must be invoked by a script in order to make the Slice types available and to configure the script's communicator. If no profile name is supplied, the default profile is loaded. A script is not allowed to load more than one profile.

For example, here is a script that loads `Profile1` shown in Section 20.3.3:

```
<?php
Ice_loadProfile("Profile1");
...
?>
```

For troubleshooting purposes, the `Ice_dumpProfile` function can be called by a script. This function displays all of the relevant information about the profile loaded by the script, including the communicator's configuration properties as well as the PHP mappings for all of the Slice definitions loaded by the profile.

20.3.5 Slice Semantics

The expense of parsing Slice files is incurred once, when the PHP interpreter initializes the IcePHP extension. For this reason, we recommend that the IcePHP extension be statically configured into the PHP interpreter, either by compiling the extension directly into the interpreter, or configuring PHP to load the extension dynamically at startup. For the same reason, we discourage the use of IcePHP in a

CGI context, in which a new PHP interpreter is created for every HTTP request. Furthermore, we strongly discourage scripts from dynamically loading the IcePHP extension using PHP's `d1` function.

20.4 Client-Side Slice-to-PHP Mapping

This section describes the PHP mapping for Slice types.

20.4.1 Mapping for Identifiers

Slice identifiers map to PHP identifiers of the same name, unless the Slice identifier conflicts with a PHP reserved word, in which case the mapped identifier is prefixed with an underscore. For example, the Slice identifier `echo` is mapped as `_echo`.

A flattened mapping is used for identifiers defined within Slice modules because PHP does not have an equivalent to C++ namespaces or Java packages. The flattened mapping uses underscores to separate the components of a fully-scoped name. For example, consider the following Slice definition:

```
module M {  
    enum E { one, two, three };  
};
```

In this case, the Slice identifier `M::E` is flattened to the PHP identifier `M_E`.

Note that when checking for a conflict with a PHP reserved word, only the fully-scoped, flattened identifier is considered. For example, the Slice identifier `M::function` is mapped as `M_function`, despite the fact that `function` is a PHP reserved word. There is no need to map it as `M__function` (with two underscores) because `M_function` does not conflict with a PHP reserved word.

However, it is still possible for the flattened mapping to generate identifiers that conflict with PHP reserved words. For instance, the Slice identifier `require::once` must be mapped as `_require_once` in order to avoid conflict with the PHP reserved word `require_once`.

20.4.2 Mapping for Simple Built-in Types

PHP has a limited set of primitive types: `boolean`, `integer`, `double`, and `string`. The Slice built-in types are mapped to PHP types as shown in Table 20.2.

Table 20.2. Mapping of Slice built-in types to PHP.

Slice	PHP
<code>bool</code>	<code>boolean</code>
<code>byte</code>	<code>integer</code>
<code>short</code>	<code>integer</code>
<code>int</code>	<code>integer</code>
<code>long</code>	<code>integer</code>
<code>float</code>	<code>double</code>
<code>double</code>	<code>double</code>
<code>string</code>	<code>string</code>

PHP's `integer` type may not accommodate the range of values supported by Slice's `long` type, therefore `long` values that are outside this range are mapped as strings. Scripts must be prepared to receive an integer or string from any operation that returns a `long` value.

20.4.3 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

Mapping for Enumerations

Enumerations map to a PHP class containing a constant definition for each enumerator. For example:

```
enum Fruit { Apple, Pear, Orange };
```

The PHP mapping is shown below:

```
class Fruit {  
    const Apple = 0;  
    const Pear = 1;  
    const Orange = 2;  
}
```

The enumerators can be accessed in PHP code as `Fruit::Apple`, etc.

Mapping for Structures

Slice structures map to PHP classes. For each Slice data member, the PHP class contains a variable of the same name. For example, here is our `Employee` structure from Section 4.7.4 yet again:

```
struct Employee {  
    long number;  
    string firstName;  
    string lastName;  
};
```

This structure is mapped to the following PHP class:

```
class Employee {  
    var $number;  
    var $firstName;  
    var $lastName;  
}
```

Mapping for Sequences

Slice sequences are mapped to native PHP indexed arrays. The first element of the Slice sequence is contained at index 0 (zero) of the PHP array, followed by the remaining elements in ascending index order.

Here is the definition of our `FruitPlatter` sequence from Section 4.7.3:

```
sequence<Fruit> FruitPlatter;
```

You can create an instance of this sequence as shown below:

```
// Make a small platter with one Apple and one Orange  
//  
$platter = array(Fruit::Apple, Fruit::Orange);
```

Mapping for Dictionaries

Slice dictionaries map to native PHP associative arrays. The PHP mapping does not currently support all Slice dictionary types, however, because native PHP associative arrays support only integer and string key types. A Slice dictionary whose key type is `boolean`, `byte`, `short`, `int` or `long` is mapped as an associative array with an integer key.² A Slice dictionary with a string key type is mapped as associative array with a string key. All other key types cause a warning to be generated.

Here is the definition of our `EmployeeMap` from Section 4.7.4:

```
dictionary<long, Employee> EmployeeMap;
```

You can create an instance of this dictionary as shown below:

```
$e1 = new Employee;  
$e1->number = 42;  
$e1->firstName = "Stan";  
$e1->lastName = "Lipmann";  
  
$e2 = new Employee;  
$e2->number = 77;  
$e2->firstName = "Herb";  
$e2->lastName = "Sutter";  
  
$em = array($e1->number => $e1, $e2->number => $e2);
```

20.4.4 Mapping for Constants

Slice constant definitions map to corresponding calls to the PHP function `define`. Here are the constant definitions we saw in Section 4.7.5:

```
const bool      AppendByDefault = true;  
const byte     LowerNibble = 0x0f;  
const string   Advice = "Don't Panic!";  
const short    TheAnswer = 42;  
const double   PI = 3.1416;  
  
enum Fruit { Apple, Pear, Orange };  
const Fruit    FavoriteFruit = Pear;
```

2. Boolean values are treated as integers, with false equivalent to 0 (zero) and true equivalent to 1 (one).

Here are the equivalent PHP definitions for these constants:

```
define("AppendByDefault", true);
define("LowerNibble", 15);
define("Advice", "Don't Panic!");
define("TheAnswer", 42);
define("PI", 3.1416);

class Fruit {
    const Apple = 0;
    const Pear = 1;
    const Orange = 2;
}
define("FavoriteFruit", Fruit::Pear);
```

20.4.5 Mapping for Exceptions

A Slice exception maps to a PHP class. For each exception member, the corresponding class contains a variable of the same name. All user exceptions ultimately derive from `Ice_UserException` (which, in turn, derives from `Ice_Exception`, which derives from PHP's base `Exception` class):

```
abstract class Ice_Exception extends Exception {
    function __construct($message = '') {
        ...
    }
}

abstract class Ice_UserException extends Ice_Exception {
    function __construct($message = '') {
        ...
    }
}
```

The optional string argument to the constructor is passed unmodified to the `Exception` constructor.

If the exception derives from a base exception, the corresponding PHP class derives from the mapped class for the base exception. Otherwise, if no base exception is specified, the corresponding class derives from `Ice_UserException`.

Here is a fragment of the Slice definition for our world time server from Section 4.8.5:

```
exception GenericError {
    string reason;
};

exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exceptions are mapped as the following PHP classes:

```
class GenericError extends Ice_UserException {
    function __construct($message = '') {
        ...
    }

    var $reason;
}

class BadTimeVal extends GenericError {
    function __construct($message = '') {
        ...
    }
}

class BadZoneName extends GenericError {
    function __construct($message = '') {
        ...
    }
}
```

An application can catch these exceptions as shown below:

```
try {
    ...
} catch(BadZoneName $ex) {
    // Handle BadZoneName
} catch(GenericError $ex) {
    // Handle GenericError
} catch(Ice_Exception $ex) {
    // Handle all other Ice exceptions
    print_r($ex);
}
```

20.4.6 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from

`Ice_LocalException` (which, in turn, derives from `Ice_Exception`, which derives from PHP's base `Exception` class):

```
abstract class Ice_Exception extends Exception {
    function __construct($message = '') {
        ...
    }

abstract class Ice_LocalException extends Ice_Exception {
    function __construct($message = '') {
        ...
    }
}
```

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 82. Note however that the PHP mapping only defines classes for the local exceptions listed below:

- `Ice_LocalException`
- `Ice_UnknownException`
- `Ice_UnknownLocalException`
- `Ice_UnknownUserException`
- `Ice_RequestFailedException`
- `Ice_ObjectNotExistException`
- `Ice_FacetNotExistException`
- `Ice_OperationNotExistException`
- `Ice_ProtocolException`
- `Ice_MarshalException`
- `Ice_NoObjectFactoryException`

Instances of all remaining local exceptions are converted to the class `Ice_UnknownLocalException`. The `unknown` member of this class contains a string representation of the original exception.

20.4.7 Mapping for Interfaces

A Slice interface maps to a PHP interface. Each operation in the interface maps to a method of the same name, as described in Section 20.4.9. The inheritance structure of the Slice interface is preserved in the PHP mapping, and all interfaces ultimately derive from `Ice_Object`:

```
interface Ice_Object {};
```

For example, consider the following Slice definitions:

```
interface A {
    void opA();
};

interface B extends A {
    void opB();
};
```

These interfaces are mapped to PHP interfaces as shown below:

```
interface A implements Ice_Object
{
    function opA();
}
interface B implements A
{
    function opB();
}
```

20.4.8 Mapping for Classes

A Slice class maps to an abstract PHP class. Each operation in the class maps to an abstract method of the same name, as described in Section 20.4.9. For each Slice data member, the PHP class contains a variable of the same name. The inheritance structure of the Slice class is preserved in the PHP mapping, and all classes ultimately derive from `Ice_ObjectImpl` (which, in turn, implements `Ice_Object`):

```
class Ice_ObjectImpl implements Ice_Object
{
}
```

Consider the following class definition:

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
    string format();      // Return time as hh:mm:ss
};
```

The PHP mapping for this class is shown below:

```
abstract class TimeOfDay extends Ice_ObjectImpl
{
    var $hour;
    var $minute;
    var $second;
    abstract function format();
}
```

Class Factories

Class factories are installed by invoking `addObjectFactory` on the communicator (see Section 20.4.11). A factory must implement the interface `Ice_ObjectFactory`, defined as follows:

```
interface Ice_ObjectFactory implements Ice_LocalObject
{
    function create(/* string */ $id);
    function destroy();
}
```

For example, we can define and install a factory for the `TimeOfDay` class as shown below:

```
class TimeOfDayI extends TimeOfDay {
    function format()
    {
        return sprintf("%02d:%02d:%02d", $this->hour,
                      $this->minute, $this->second);
    }
}

class TimeOfDayFactory extends Ice_LocalObjectImpl
    implements Ice_ObjectFactory {
    function create($id)
    {
        return new TimeOfDayI;
    }

    function destroy() {}
}

$ICE->addObjectFactory(new TimeOfDayFactory, "::__TimeOfDay");
```

20.4.9 Mapping for Operations

Each operation defined in a Slice class or interface is mapped to a PHP function of the same name. Furthermore, each parameter of an operation is mapped to a PHP parameter of the same name, with `out` parameters passed by reference. Since PHP is a loosely-typed language, no parameter types are specified.³

Consider the following interface:

```
interface I {
    float op(string s, out int i);
};
```

The PHP mapping for this interface is shown below:

```
interface I {
    function op($s, &$i);
}
```

20.4.10 Mapping for Proxies

The PHP mapping for Slice proxies uses a single interface, `Ice_ObjectPrx`, to represent all proxy types.

Typed vs. Untyped Proxies

A proxy can be typed or untyped. All proxies, whether they are typed or untyped, support the core proxy methods described in the next section.

An *untyped proxy* is equivalent to the Slice type `Object*`. The communicator operation `stringToProxy` returns an untyped proxy, as do several of the core proxy methods. A script cannot invoke user-defined operations on an untyped proxy, nor can an untyped proxy be passed as an argument where a typed proxy is expected.

A *typed proxy* is one that has been associated with a Slice class or interface type. There are two ways a script can obtain a typed proxy:

1. By receiving it as the result of a remote invocation that returns a typed proxy.

3. PHP5 introduces the notion of “type hints” that allow you to specify the formal type of object parameters. This would enable the Slice mapping to specify type hints for parameters of type struct, interface, class and proxy. Unfortunately, PHP5 does not currently allow a parameter defined with a type hint to receive a null value, therefore the Slice mapping does not use type hints for parameters.

2. By using the core proxy methods `ice_checkedCast` or `ice_uncheckedCast`.

For example, suppose our script needs to obtain a typed proxy for interface A, shown below:

```
interface A {  
    void opA();  
};
```

Here are the steps our script performs:

```
$obj = $ICE->stringToProxy("a:tcp -p 12345");  
$obj->opA(); // WRONG!  
$a = $obj->ice_checkedCast("::A");  
$a->opA(); // OK
```

Attempting to invoke `opA` on `$obj` would result in a fatal error, because `$obj` is an untyped proxy.

Core Proxy Methods

The `Ice_ObjectPrx` interface provides a number of core proxy methods:

```
interface Ice_ObjectPrx {  
    function ice_isA(/* string */ $id, /* array */ $ctx = null);  
    function ice_ping(/* array */ $ctx = null);  
    function ice_ids(/* array */ $ctx = null);  
    function ice_id(/* array */ $ctx = null);  
    function ice_getIdentity();  
    function ice_newIdentity(/* Ice_Identity */ $id);  
    function ice_getContext();  
    function ice_newContext(/* array */ $ctx);  
    function ice_getFacet();  
    function ice_newFacet(/* string */ $facet);  
    function ice_twoway();  
    function ice_isTwoway();  
    function ice_oneway();  
    function ice_isOneway();  
    function ice_batchOneway();  
    function ice_isBatchOneway();  
    function ice_datagram();  
    function ice_isDatagram();  
    function ice_batchDatagram();  
    function ice_isBatchDatagram();  
    function ice_secure(/* boolean */ $b);  
    function ice_compress(/* boolean */ $b);  
    function ice_timeout(/* integer */ $t);
```

```
function ice_default();
function ice_uncheckedCast(/* string */ $type,
                         /* string */ $facet = null);
function ice_checkedCast(/* string */ $type,
                         /* string */ $facet = null);
}
```

These methods can be categorized as follows:

- Remote inspection: methods that return information about the remote object.
- Local inspection: methods that return information about the proxy's local configuration.
- Factory: methods that return new proxy instances configured with different features.

Proxies are immutable, so factory methods allow an application to obtain a new proxy with the desired configuration. Factory methods essentially clone the original proxy and modify one or more features of the new proxy.

To demonstrate the use of proxy factory methods, consider this example:

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_oneway();
$p = $p->ice_secure(true);
$p = $p->ice_uncheckedCast(":A");
```

The script receives an untyped proxy as the return value of `stringToProxy`. Since the stringified proxy did not contain any proxy options, it is configured by default as a twoway, insecure proxy with no timeout. However, our goal is to obtain a secure oneway proxy for interface A, therefore we invoke `ice_oneway`, followed by `ice_secure`. At this point, we have an untyped proxy configured for secure oneway invocations. Finally, we call `ice_uncheckedCast` to obtain a typed proxy.⁴

Note that this process can be simplified by chaining the proxy factory methods, as shown below:

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_oneway()->ice_secure(true)->ice_uncheckedCast(":A");
```

The order that the proxy factory methods are invoked is usually not important, except in the case of `ice_checkedCast` and `ice_uncheckedCast`. For

4. We cannot use `ice_checkedCast` on a proxy configured for oneway invocations.

example, if the script above had invoked `ice_uncheckedCast` first, followed by the other factory methods, then the result would have been an untyped proxy:

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_uncheckedCast("::A"); // WRONG!
$p = $p->ice_oneway();
$p = $p->ice_secure(true);
```

The reason for this unexpected behavior is that most of the factory methods return an untyped proxy, thereby discarding any type that might have been associated with the original proxy. By invoking `ice_oneway` on the typed proxy returned from `ice_uncheckedCast`, the script has lost its typed proxy.

The core proxy methods are explained in greater detail in Table 20.3.

Table 20.3. Description of core proxy methods.

Method	Description	Remote
<code>ice_isA</code>	Returns <code>true</code> if the remote object supports the type indicated by the <code>id</code> argument, otherwise <code>false</code> . This method can only be invoked on a twoway proxy.	Yes
<code>ice_ping</code>	Determines whether the remote object is reachable. Does not return a value.	Yes
<code>ice_ids</code>	Returns the type ids of the types supported by the remote object. The return value is an array of strings. This method can only be invoked on a twoway proxy.	Yes
<code>ice_id</code>	Returns the type id of the most-derived type supported by the remote object. This method can only be invoked on a twoway proxy.	Yes
<code>ice_getIdentity</code>	Returns the Ice object identity associated with the proxy. The return value is an instance of <code>Ice_Identity</code> .	No
<code>ice_newIdentity</code>	Returns a new untyped proxy having the given identity.	No
<code>ice_getContext</code>	Returns the request context associated with the proxy. The return value is an associative array representing the mapping for <code>Ice::Context</code> .	No
<code>ice_newContext</code>	Returns a new untyped proxy having the given request context.	No

Table 20.3. Description of core proxy methods.

Method	Description	Remote
<code>ice_getFacet</code>	Returns the name of the facet associated with the proxy, or an empty string if no facet has been set.	No
<code>ice_newFacet</code>	Returns a new untyped proxy having the given facet name.	No
<code>ice_twoway</code>	Returns a new untyped proxy for making twoway invocations.	No
<code>ice_isTwoway</code>	Returns <code>true</code> if the proxy uses twoway invocations, otherwise <code>false</code> .	No
<code>ice_oneway</code>	Returns a new untyped proxy for making oneway invocations.	No
<code>ice_isOneway</code>	Returns <code>true</code> if the proxy uses oneway invocations, otherwise <code>false</code> .	No
<code>ice_batchOneway</code>	Returns a new untyped proxy for making batch oneway invocations.	No
<code>ice_isBatchOneway</code>	Returns <code>true</code> if the proxy uses batch oneway invocations, otherwise <code>false</code> .	No
<code>ice_datagram</code>	Returns a new untyped proxy for making datagram invocations.	No
<code>ice_isDatagram</code>	Returns <code>true</code> if the proxy uses datagram invocations, otherwise <code>false</code> .	No
<code>ice_batchDatagram</code>	Returns a new untyped proxy for making batch datagram invocations.	No
<code>ice_isBatchDatagram</code>	Returns <code>true</code> if the proxy uses batch datagram invocations, otherwise <code>false</code> .	No
<code>ice_secure</code>	Returns a new untyped proxy whose endpoints may be filtered depending on the boolean argument. If <code>true</code> , only endpoints using secure transports are allowed, otherwise all endpoints are allowed.	No
<code>ice_compress</code>	Returns a new untyped proxy whose protocol compression capability is determined by the boolean argument. If <code>true</code> , the proxy uses protocol compression if it is supported by the endpoint. If <code>false</code> , protocol compression is never used.	No

Table 20.3. Description of core proxy methods.

Method	Description	Remote
<code>ice_timeout</code>	Returns a new untyped proxy with the given timeout value in milliseconds. A value of <code>-1</code> disables timeouts.	No
<code>ice_default</code>	Returns a new untyped proxy having the default proxy configuration.	No
<code>ice_uncheckedCast</code>	Returns a new typed proxy associated with the given type id. The Slice definition for the type id must be available in the current profile. If a facet name is provided, the returned proxy is configured with the facet.	No
<code>ice_checkedCast</code>	If the remote object supports the type indicated by the given type id, this method returns a new typed proxy associated with the type, otherwise it returns null. The Slice definition for the type id must be available in the current profile. If a facet name is provided, the returned proxy is configured with the facet.	Yes

Request Context

All remote operations on a proxy support an optional final parameter of type `Ice::Context` representing the request context (see Section 16.8). The standard PHP mapping for the request context is an associative array in which the keys and values are strings. For example, the code below illustrates how to invoke `ice_ping` with a request context:

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$ctx = array("theKey" => "theValue");
$p->ice_ping($ctx);
```

Alternatively, a script can specify a default request context using the proxy method `ice_newContext`. See Section 16.8.2 for more information.

Identity

Certain core proxy operations use the type `Ice_Identity`, which is the PHP mapping for the Slice type `Ice::Identity`. This type is mapped using the standard rules for Slice structures, therefore it is defined as follows:

```
class Ice_Identity {  
    var $name;  
    var $category;  
}
```

Two global functions are provided for converting `Ice_Identity` values to and from a string representation:

```
function Ice_stringToIdentity(/* string */ $str);  
function Ice_identityToString(/* Ice_Identity */ $id);
```

20.4.11 Mapping for `Ice::Communicator`

Since the Ice extension for PHP provides only client-side facilities, many of the operations provided by `Ice::Communicator` operations are not relevant, therefore the PHP mapping supports a subset of the communicator operations. The mapping for `Ice::Communicator` is shown below:

```
interface Ice_Communicator {  
    function stringToProxy(/* string */ $str);  
    function proxyToString(/* Ice_ObjectPrx */ $prx);  
    function addObjectFactory(/* Ice_ObjectFactory */ $factory,  
        /* string */ $id);  
    function removeObjectFactory(/* string */ $id);  
    function findObjectFactory(/* string */ $id);  
    function flushBatchRequests();  
}
```

See Appendix B for a description of these operations.

Communicator Lifecycle

PHP scripts are not allowed to create or destroy communicators. Rather, a communicator is created prior to each PHP request, and is destroyed after the request completes.

Accessing the Communicator

The communicator instance created for a request is available to the script via the global variable `$ICE`. Scripts should not attempt to assign a different value to this variable. As with any global variable, scripts that need to use `$ICE` from within a function must declare the variable as global:

```
function printProxy($prx) {
    global $ICE;

    print $ICE->proxyToString($prx);
}
```

Communicator Configuration

The profile loaded by the script determines the communicator's configuration. See Section 20.3 for more information.

Part IV

Ice Services

Chapter 21

IcePack

21.1 Chapter Overview

In this chapter we present IcePack. Section 21.2 provides a brief introduction to IcePack’s features, and Sections 21.3 and 21.4 review some concepts which are helpful in understanding how IcePack works. The purpose and use of the IcePack components are described in Sections 21.5 through 21.7. Deploying applications with IcePack is discussed in Sections 21.8 and 21.9, and some troubleshooting tips are provided in Section 21.10.

21.2 Introduction

IcePack provides several essential services that simplify the development and deployment of a complex distributed application:

- a locator service to locate objects and ensure location transparency,
- a server activation and monitoring service to manage server processes,
- and a powerful server deployment mechanism.

IcePack is comprised of two main components:

- An IcePack *registry* manages all of the information about the applications deployed in a particular domain.

- An IcePack *node* activates and monitors server processes.

There is only one IcePack registry in a domain, but there can be many IcePack nodes (typically one per host).

21.3 Concepts

Throughout most of this book we have presented examples that intentionally avoid (for the sake of simplicity) some important issues of distributed application development, including location transparency, binding, and deployment. IcePack addresses these issues with minimal impact on Ice applications.

21.3.1 Location Transparency

Location transparency means the location of an object implementation is irrelevant to the client; the object could be implemented within the client's process, or in a process on another host. Regardless of where the object is implemented, the client interacts with the object in exactly the same way, and with the same semantics. Location transparency simplifies the development of distributed applications and is the foundation for the services that IcePack provides.

21.3.2 Binding

As described in Section 2.2, binding is the process of connecting a proxy with its servant. In practice, this typically involves opening a socket connection to the server's endpoint. However, there are two binding modes that the Ice run time can use to obtain the server's endpoint information: direct binding and indirect binding.

Direct Binding

We have seen endpoint details repeated many times in the examples in this book: the server calls `createObjectAdapterWithEndpoints`, specifying a protocol and port number, and the client invokes `stringToProxy` on a stringified proxy containing the same protocol and port number. The Ice run time in the client opens a socket directly to the server's endpoint; if the server is not currently running, or is running on a different host or port, then the client receives an exception. This is known as *direct binding*, and while it is easy to understand and quite convenient, it is not appropriate for many enterprise applications.

We can improve this situation somewhat by externalizing the endpoint details using configuration properties (see Appendix C). For example, the endpoints of an object adapter named `MyAdapter` can be specified using a configuration property such as:

```
MyAdapter.Endpoints=tcp -p 9999
```

Similarly, the client can define a property containing the stringified proxy:

```
MyProxy=theObject:tcp -p 9999
```

This is a step in the right direction; at least the endpoint details are no longer contained in the code. The server can now be relocated to a different host or port without requiring changes to either client or server programs.

This is still not an ideal solution, however. For example, if proxies were stored in a database, they would all have to be updated when the server endpoints change. Similarly, if there are multiple client and server applications running on multiple hosts, the administrative work involved in updating configuration files to reflect evolving server configurations can quickly become unwieldy.

Indirect Binding

As its name implies, *indirect binding* adds a layer of indirection such that the client no longer needs to specify endpoint details for a proxy. Rather, a proxy's endpoints are determined dynamically by the Ice run time. We discuss how Ice makes this possible in Section 21.4, but for now we can illustrate its use with more proxy examples.

First, we replace the endpoint details with an object adapter identifier:

```
MyProxy=theObject@theAdapter
```

The symbol `theAdapter` identifies an object adapter whose endpoints are retrieved automatically.

Next, we can use only the object identity:

```
MyProxy=theObject
```

In this case, the object identity alone is sufficient to locate an object.

21.3.3 Deployment

Application deployment is a broad term with plenty of application-specific requirements. With respect to Ice applications, this can involve the creation and installation of configuration files, initializing IceBox services (see Chapter 26), and writing scripts to handle startup and shutdown procedures. IcePack simplifies

these tasks using an administrative tool that processes XML descriptions of your application's services and deploys them automatically.

21.4 Ice Locator Facility

The Ice run time supports a delegation facility, represented by the `Ice::Locator` interface, that allows endpoint information to be obtained dynamically. This facility enables the flexibility of indirect binding described in Section 21.3.2 and is a fundamental component of the services provided by IcePack. The details of the `Ice::Locator` interface definition are not relevant to this discussion, but its capabilities are worth exploring in order to understand how IcePack simplifies the design and development of Ice applications.

21.4.1 Objects and Object Adapters

Recall from Section 21.3.2 the two stringified proxy examples that demonstrate indirect binding:

```
MyProxy=theObject@theAdapter
```

and

```
MyProxy=theObject
```

These proxies show the two methods of obtaining endpoint information: by object adapter identifier, and by object identity. In either case, the Ice run time in the client obtains the endpoint information by making requests on an `Ice::Locator` object. This activity is transparent to the application and occurs whenever a proxy using indirect binding is created, such as by calling `stringToProxy` or when receiving a proxy as the result of an operation. However, the Ice run time is careful to minimize the number of locator requests (see Section 21.4.7).

A locator is essentially a repository containing two lookup tables: one that associates endpoints with object identities, and another that associates endpoints with object adapter identifiers. Figure 21.1 illustrates the role played by a locator:

1. At startup, the server registers its well-known objects and object adapters with the locator.
2. When creating a proxy using indirect binding, the Ice run time in the client invokes on the locator to obtain the endpoints associated with an object identity or object adapter identifier.

3. After obtaining the endpoints for the proxy, the client establishes a connection directly to the server. The locator is not consulted again for that proxy unless a connection error occurs.

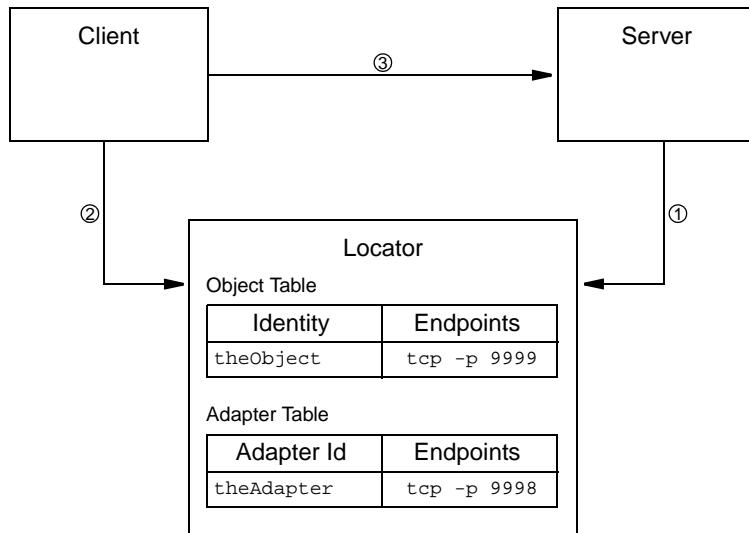


Figure 21.1. Obtaining endpoint information from a locator.

21.4.2 Design Considerations

Although both proxy styles shown in Section 21.4.1 are syntactically similar in that they eliminate the dependency on endpoint details, they have distinctly different design considerations. Specifically, the object identity style is intended to be used primarily for bootstrapping purposes; i.e., the identities of a few well-known objects are made available via the locator facility, and clients use these well-known objects to obtain proxies for other objects as necessary.

The object adapter style, on the other hand, scales to many objects but is still dependent on the server configuration (i.e., the object adapter identifier).

The best situation is often a combination of the two methods: the client uses the object identity style to bootstrap a principal object that provides proxies to subordinate objects using the object adapter style.

21.4.3 Using the Ice Locator Facility in Clients

Integrating the locator functionality into a client application is merely a matter of configuration: the value of the property `Ice.Default.Locator` must specify a stringified proxy for the locator object. Once the Ice run time has this proxy, it can convert object adapter identifiers and object identities into physical endpoint attributes. See Section 21.5.4 for more information on defining the locator property.

21.4.4 Using the Ice Locator Facility in Servers

An object adapter automatically registers its endpoints with the locator when the adapter is activated. Automatic registration is enabled by configuring the object adapter with an identifier¹ and a locator proxy. Of course, the object adapter must also have one or more endpoints defined, but this requirement is not specific to the locator.

For example, an object adapter named `MyAdapter` could be configured with the following properties:

```
Ice.Default.Locator=... // the stringified locator proxy  
MyAdapter.AdapterId=theAdapter  
MyAdapter.Endpoints=default
```

This configuration assigns the locator identifier `theAdapter` to the object adapter named `MyAdapter`. The endpoint for this object adapter will use the default protocol and a system-assigned port number. Upon activation, the object adapter registers this endpoint with the locator whose proxy is provided in `Ice.Default.Locator`, thereby enabling clients to use the object adapter proxy style with this adapter.

As its name implies, the `Ice.Default.Locator` property supplies the default locator proxy for all object adapters in a server. However, each object adapter whose endpoints should be registered with the locator must also be configured with an identifier. A server can therefore create as many object adapters as necessary, with complete control over which ones are registered with a locator.

The locator proxy can also be configured for a particular object adapter using a property such as:

1. There is a potential for conflicts if object adapters in a domain do not use unique identifiers; it is the developer's responsibility to ensure that this does not occur.

```
MyAdapter.Locator=... // the stringified locator proxy
```

This property has precedence over the `Ice.Default.Locator` property. See Section 21.5.4 for more information on defining the locator properties.

To facilitate use of the proxy style of object identity, each object's identity and proxy must be registered. This can be done programmatically by the server, administratively using the IcePack command-line tool described in Section 21.7, or automatically using the IcePack deployment mechanism discussed in Section 21.8.

21.4.5 Deployment

The locator facility simplifies application deployment, because we no longer need to bother assigning unique endpoints to object adapters. Since proxies contain only a symbolic reference to the endpoint information, object adapters are free to use system-assigned (as opposed to user-assigned) ports.

Although an object adapter would likely be assigned a different port each time it is activated, its automatic registration with the locator ensures that clients receive the correct endpoint information. If a client holds a proxy to an object in an object adapter that is deactivated and then reactivated on a different port, the retry behavior in the client's Ice run time will automatically refresh the proxy's endpoints.

21.4.6 Dependencies

The locator facility is represented by `Ice::Locator`, a generic Slice interface that can have multiple implementations. When the Ice run time is configured with an `Ice::Locator` proxy, it automatically uses the locator to perform indirect binding. The Ice run time therefore has a dependency on the `Ice::Locator` interface, but not on IcePack; the IcePack registry simply implements the `Ice::Locator` interface, thus allowing clients to use indirect binding to servers that are activated by IcePack nodes.

21.4.7 Overhead of Indirect Binding

As you might expect, indirect binding does create some additional overhead, since the Ice run time must make a proxy invocation on the locator to obtain endpoint information. To minimize the number of locator invocations, the Ice run time caches the results; subsequent attempts to bind to the same identity or object

adapter name will use the cached endpoint information. For the majority of applications, the flexibility provided by indirect binding far outweighs the minimal overhead incurred by its use.

21.5 IcePack Registry

The IcePack registry is a centralized repository of information, including

- IcePack nodes,
- deployed servers and object adapters,
- object registrations.

The registry also implements the `Ice::Locator` interface described in Section 21.4, and supports additional Slice interfaces for querying and administrative tasks. A registry service can optionally be collocated with an IcePack node, which conserves resources and can be convenient during development and testing.

21.5.1 Endpoints

The IcePack registry creates up to four sets of endpoints, configured with the following properties:

- `IcePack.Registry.Client.Endpoints`
Client-side endpoints supporting the `Ice::Locator` and `IcePack::Query` interfaces.
- `IcePack.Registry.Server.Endpoints`
Server-side endpoints for object and object adapter registration.
- `IcePack.Registry.Admin.Endpoints`
Administrative endpoints supporting the `IcePack::Admin` interface (optional).
- `IcePack.Registry.Internal.Endpoints`
Internal endpoints used by IcePack nodes to communicate with the registry (optional). This property is optional only if the registry is being used without any nodes.

See Appendix C for more information on these properties.

21.5.2 Security Considerations

Access to an IcePack registry's administrative interface should be restricted if the registry is running on an insecure host, such as an application firewall. The `IcePack.Registry.Admin.Endpoints` property (described in the previous section) offers several alternatives:

- Disable administrative access completely by not defining the property.
- Block external access to the administrative endpoints defined by the property.
- Use only SSL endpoints in the property (see Chapter 24).

The method you choose depends on your deployment requirements. For example, if no administrative access is necessary, then the first technique is the most secure. If that is not feasible, then physically blocking (via a firewall) external access to the administrative endpoints is recommended. Finally, if a registry must be administered externally, then SSL endpoints should be used to ensure that only authenticated clients gain access.

21.5.3 Registry Object Identities

The identities of the primary IcePack registry objects are shown below:

- `IcePack/Locator` for the `Ice::Locator` implementation.
- `IcePack/Query` for the `IcePack::Query` implementation.
- `IcePack/Admin` for the `IcePack::Admin` implementation.

These identities, together with the endpoints defined in the properties from Section 21.5.1, enable you to construct proxies for the registry objects.

21.5.4 Ice Locator Properties

Now that we know the identity of the registry's `Ice::Locator` implementation, and the endpoints defined in the `IcePack.Registry.Client.Endpoints` property, we can define the configuration property for Ice clients and servers that enables the locator functionality.

In general, clients and servers can both use the `Ice.Default.Locator` property. For example, if the IcePack registry client endpoint is `tcp -p 12000`, then the locator property can be defined as follows:

```
Ice.Default.Locator=IcePack/Locator:tcp -p 12000
```

In servers, a locator proxy can also be specified individually for each object adapter using a property definition such as:

```
MyAdapter.Locator=IcePack/Locator:tcp -p 12000
```

If the locator property is not defined for an adapter, the value of `Ice.Default.Locator` will be used. See Appendix C for more information on these properties.

21.5.5 Usage

The IcePack registry supports the following command-line options:

```
$ icepackregistry -h
Usage: icepackregistry [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
--nowarn            Don't print any security warnings.
```

The registry can optionally be run as a Unix daemon or Win32 service. On Unix, the following additional commands are supported:

```
--daemon           Run as a daemon.
--noclose          Do not close open file descriptors.
--nochdir          Do not change the current working directory.
```

On Windows, the following additional commands are supported:

```
--service NAME      Run as the Windows service NAME.

--install NAME [--display DISP] [--executable EXEC] [args]
                         Install as Windows service NAME. If DISP is
                         provided, use it as the display name,
                         otherwise NAME is used. If EXEC is provided,
                         use it as the service executable, otherwise
                         this executable is used. Any additional
                         arguments are passed unchanged to the
                         service at startup.

--uninstall NAME     Uninstall Windows service NAME.

--start NAME [args]   Start Windows service NAME. Any additional
                         arguments are passed unchanged to the
                         service.

--stop NAME          Stop Windows service NAME.
```

Please see Section 10.3.2 for more information on these options.

21.5.6 Configuration

IcePack registry configuration properties have the prefix `IcePack.Registry` and are described in Appendix C.

21.5.7 Location Service

An IcePack registry can be used by itself as a location service for applications that need indirect binding but do not require the server activation features of an IcePack node.

For example, the following configuration file defines the minimal set of properties that the registry requires to operate in this mode:

```
IcePack.Registry.Client.Endpoints=default -p 12000
IcePack.Registry.Server.Endpoints=default
IcePack.Registry.Admin.Endpoints=default
IcePack.Registry.Data=db
IcePack.Registry.DynamicRegistration=1
```

Since no IcePack nodes will be running with this registry, it is not necessary to define the `IcePack.Registry.Internal.Endpoints` property. The `IcePack.Registry.DynamicRegistration` property is required to allow servers to dynamically register their object adapters without having previously configured them via deployment.

We can initialize the registry's database directory and start it using the following commands:

```
$ mkdir db
$ icepackregistry --Ice.Config=config
```

The last command assumes the configuration file exists in the current directory with the name `config`.

Clients and servers can now use this registry by defining the default locator property using the endpoint in the registry's configuration file:

```
Ice.Default.Locator=IcePack/Locator:default -p 12000
```

A manually-activated server that is configured with this property will register each of its object adapters that has an `AdapterId` property, allowing clients to use the object adapter proxy style. Furthermore, you can use the administration tool (see Section 21.7) to advertise well-known objects in the registry and allow clients to use the object proxy style.

21.6 IcePack Node

An IcePack node is a process that activates, monitors, and deactivates registered server processes. Servers are registered with IcePack nodes via the IcePack deployment mechanism described in Section 21.8.

You can run any number of IcePack nodes in a domain, but typically there is one node per host. An IcePack node must be running on each host on which servers are activated automatically, and nodes cannot run without an IcePack registry.

21.6.1 Server Activation

On-demand server activation is a valuable feature of distributed computing architectures for a number of reasons:

- it minimizes application startup times by avoiding the need to pre-start all servers;
- it allows administrators to use their computing resources more efficiently because only those servers that are actually needed are running;
- it provides more reliability in the case of some server failure scenarios, e.g., the server is reactivated after a failure and may still be capable of providing some services to clients until the failure is resolved;
- it allows remote activation and deactivation.

Activation in Detail

Activation occurs when an Ice client requests the endpoints of one of the server's object adapters via the Ice locator facility (see Section 21.4). If the server is not active at the time the client issues the request, the node activates the server and waits for the server's object adapter to register its endpoints. Once the object adapter endpoints are registered, the node sends the endpoint information back to the client. This sequence ensures that the client receives the endpoint information *after* the server is ready to receive requests.

Requirements

In order to use on-demand activation for an object adapter, the adapter must have an identifier and be entered in the IcePack registry.

Efficiency

One of the advantages of on-demand activation listed above is the ability to manage computing resources more efficiently. Of course there are many aspects to this, but Ice makes one technique particularly simple: servers can be configured to terminate gracefully after they have been idle for a certain amount of time.

A typical scenario involves a server that is activated on demand, used for a while by one or more clients, and then terminated automatically when no requests have been made for a configurable number of seconds. All that is necessary is setting the configuration property `Ice.ServerIdleTime` to the desired idle time.

21.6.2 Endpoint Registration

Section 21.4.4 discusses the configuration requirements for enabling automatic endpoint registration in servers. It should be noted however that IcePack simplifies the configuration process in two ways:

1. A server that is activated automatically by an IcePack node does not need to explicitly configure a proxy for the locator because the IcePack node defines it on the server's command-line.
2. The IcePack deployment mechanism automates the creation of a configuration file for the server, including the definition of object adapter identifiers and endpoints (see Section 21.8.5).

21.6.3 Server Registration and Deactivation

After activating a server, the IcePack node does not normally deactivate the server unless it is explicitly requested via an administrative command. The IcePack node first attempts to deactivate the server gracefully, and then waits for the server to perform an orderly shutdown. If the server does not terminate on its own after a certain period of time, the IcePack node terminates the process.

There are two ways for an IcePack node to request an orderly shutdown of a server:

1. By invoking on the proxy of an `Ice::Process` object residing in the server.
2. Using platform-specific mechanisms, such as POSIX signals.

The latter works well on POSIX platforms but requires the server to be prepared to intercept signals (see Section 15.11). On Windows platforms, it works less reliably for C++ servers, and not at all for Java servers. For these reasons, the former

alternative is preferred because it is portable and reliable, and an IcePack node only uses the latter alternative if no `Ice::Process` proxy is supplied.

The simplest and most-common way for a server to register an `Ice::Process` proxy is to set the object adapter property `RegisterProcess` to a non-zero value. This setting causes a newly-activated object adapter to create a servant implementing the `Ice::Process` interface, add that servant to the active servant map using a UUID for the identity, and invoke an operation on the Ice locator facility to register the proxy.

The object adapter also requires a value for the `Ice.ServerId` property, which should be set to the server's name. The IcePack node defines this property for automatically-activated servers.

A server should configure at most one of its object adapters with the `RegisterProcess` property. When using IcePack deployment descriptors, the `RegisterProcess` property can be defined in the server's configuration automatically by setting the `register` attribute of the `adapter` element to true (see Section 21.9).

The `Ice::Process` servant responds to the IcePack node's deactivation request by invoking `shutdown` on the communicator. This presents a potential security hole: if a hostile client obtained the proxy, it could make a denial-of-service attack by shutting down the server. It is therefore important that careful consideration be given to the server's endpoints in order to minimize the risk of an attack. Specifically, an object adapter can be configured with secure endpoints, or a separate object adapter can be dedicated to the `Ice::Process` servant and configured with an endpoint that is accessible only to the IcePack node.

21.6.4 Usage

The IcePack node supports the following command-line options:

```
$ icepacknode -h
Usage: icepacknode [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
--nowarn            Don't print any security warnings.

--deploy DESCRIPTOR [TARGET1 [TARGET2 ...]]
                    Deploy descriptor in file DESCRIPTOR, with
                    optional targets.
```

The `--deploy` option allows an application to be deployed automatically as the node process starts, which can be especially useful during testing. The name of a descriptor file should be given, and optionally the names of individual targets within the descriptor file.

The node can optionally be run as a Unix daemon or Win32 service. On Unix, the following additional commands are supported:

<code>--daemon</code>	Run as a daemon.
<code>--noclose</code>	Do not close open file descriptors.
<code>--nochdir</code>	Do not change the current working directory.

On Windows, the following additional commands are supported:

<code>--service NAME</code>	Run as the Windows service NAME.
<code>--install NAME [--display DISP] [--executable EXEC] [args]</code>	Install as Windows service NAME. If DISP is provided, use it as the display name, otherwise NAME is used. If EXEC is provided, use it as the service executable, otherwise this executable is used. Any additional arguments are passed unchanged to the service at startup.
<code>--uninstall NAME</code>	Uninstall Windows service NAME.
<code>--start NAME [args]</code>	Start Windows service NAME. Any additional arguments are passed unchanged to the service.
<code>--stop NAME</code>	Stop Windows service NAME.

Please see Section 10.3.2 for more information on these options.

21.6.5 Configuration

IcePack node configuration properties have the prefix `IcePack.Node` and are described in Appendix C.

21.7 IcePack Administration Tool

The IcePack administration tool is a command-line program that provides administrative control over all aspects of IcePack operation, including the deployment of applications and servers, and querying for information about registered entities.

The tool requires that the `Ice.Default.Locator` property be specified as described in Section 21.5.4.

21.7.1 Usage

The IcePack administration tool supports the following command-line options:

```
$ icepackadmin -h
Usage: icepackadmin [options] [file...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
-DNAME              Define NAME as 1.
-DNAME=DEF          Define NAME as DEF.
-UNAME              Remove any definition for NAME.
-IDIR               Put DIR in the include file search path.
-e COMMANDS         Execute COMMANDS.
-d, --debug          Print debug messages.
```

The `-e` option causes the tool to execute the given commands and then exit without entering an interactive session. Otherwise, the tool enters an interactive session; the `help` command displays the following usage information:

help

Print this message.

exit, quit

Exit this program.

application add DESC [TARGET1 [TARGET2 ...]]

Add servers described in application descriptor DESC. If specified the optional targets will be deployed.

application remove DESC

Remove servers described in application descriptor DESC.

node list

List all registered nodes.

node ping NAME

Ping node NAME.

```
node shutdown NAME
    Shutdown node NAME.

server list
    List all registered servers.

server add NODE NAME DESC [PATH [LIBPATH [TARGET1
[TARGET2 ...]]]]
    Add server NAME to node NODE with deployment descriptor DESC, optional
    PATH and LIBPATH. If specified the optional targets will be deployed.

server remove NAME
    Remove server NAME.

server describe NAME
    Get server NAME description.

server state NAME
    Get server NAME state.

server pid NAME
    Get server NAME pid.

server start NAME
    Start server NAME.

server stop NAME
    Stop server NAME.

server signal NAME SIGNAL
    Send SIGNAL (e.g. SIGTERM or 15) to server NAME.

server stdout NAME MESSAGE
    Write MESSAGE on server NAME's stdout.

server stderr NAME MESSAGE
    Write MESSAGE on server NAME's stderr.

server activation NAME [on-demand | manual]
    Set the server activation mode to on-demand or manual activation.
```

adapter list

List all registered adapters.

adapter endpoints NAME

Get adapter NAME endpoints.

object add PROXY [TYPE]

Add an object to the object registry, optionally specifies its type.

object remove IDENTITY

Remove an object from the object registry.

object find TYPE

Find all objects with the type TYPE.

shutdown

Shut the IcePack registry down.

21.8 Deployment

Deployment, with respect to IcePack, means configuring an IcePack domain for an application. Specifically, deployment involves adding well-known objects and object adapters to the IcePack registry, configuring servers on IcePack nodes, and creating server configuration files.

IcePack provides a data-driven deployment service in which *descriptors* are written in XML and deployed as often as necessary. There are three types of deployment descriptors:

- Application descriptors define servers to be deployed, and the nodes on which to deploy them.
- Server descriptors define Ice server applications written in C++ or Java. Server descriptors can also define IceBox servers (see Chapter 26).
- Service descriptors define IceBox services.

During deployment, IcePack configures its various components as directed by the descriptors. Each descriptor must reside in its own file.

21.8.1 Application Descriptors

With respect to IcePack, an *application* is defined as a collection of servers and/or IceBox services running on one or more IcePack nodes. An application descriptor is structured similarly, as illustrated by the following example:

```
<application>
    <node name="Firewall" basedir="${basedir}">
        <server name="AppGateway" binpath="./appgateway"
               descriptor="appgateway.xml"/>
    </node>
    <node name="Backend" basedir="${basedir}">
        <server name="Account" binpath="/opt/Ice/bin/icebox"
               descriptor="account.xml"/>
    </node>
</application>
```

Inside the outer `application` element are two `node` elements defining IcePack nodes: `Firewall` and `Backend`. The server `AppGateway` will be deployed on `Firewall`, and the server `Account` will be deployed on `Backend`. Paths to the executables, as well as server descriptor files, are supplied in the attributes of each `server` element.

21.8.2 Server Descriptors

A server descriptor defines a server's object adapters, well-known objects, and configuration properties.

Continuing our example from the preceding section, the contents of `appgateway.xml` are shown below:

```
<server kind="cpp">
    <adapters>
        <adapter name="GatewayAdapter" endpoints="default">
            <object identity="${name}" type="::AppGateway"/>
        </adapter>
    </adapters>

    <properties>
        <property name="Account.Proxy" value="AccountManager"/>
    </properties>
</server>
```

This server descriptor defines a C++ server with one object adapter named `GatewayAdapter` with a default endpoint configuration. The adapter defines a well-known object of type `::AppGateway` and an identity with the value of the vari-

able \${name}. In this example, \${name} is substituted with AppGateway, which is the name of the server in the application descriptor.

The server descriptor also defines a property named Account.Proxy having the value AccountManager, representing a proxy for a well-known object that will be resolved via the Ice locator facility.

The descriptor in account.xml is shown below:

```
<server kind="cpp-icebox" endpoints="default">
    <service name="AccountService" descriptor="accountsrv.xml" />
</server>
```

This server descriptor defines a C++ IceBox server with a default endpoint configuration. The server is configured with an IceBox service named AccountService, whose descriptor is located in the file accountsvc.xml.

For more information on IceBox, see Chapter 26.

21.8.3 Service Descriptors

A service descriptor defines an IceBox service, including its object adapters, well-known objects, and configuration properties.

Continuing our example from the preceding sections, here are the contents of accountsvc.xml:

```
<service kind="standard" entry="AccountService:create">
    <adapters>
        <adapter name="AccountAdapter" endpoints="default">
            <object identity="AccountManager"
                type="::AccountManager" />
        </adapter>
    </adapters>
</service>
```

This service descriptor defines an IceBox service with one object adapter named AccountAdapter with a default endpoint configuration. The adapter defines a well-known object of type ::AccountManager and the identity AccountManager. This is the well-known object referred to by the AppGateway server descriptor in Section 21.8.2.

21.8.4 Requirements

To deploy an application, the IcePack registry must be running, and all IcePack nodes on which servers will be deployed must be running.

21.8.5 Configuring Servers and Services

IcePack generates a configuration file for each deployed server and service. The file contains the following properties:

- AdapterId and Endpoints properties for each adapter element
- RegisterProcess property for a server's adapter, if necessary
- an equivalent definition for each property element
- an IceBox.DBEnvName.Name property for a Freeze service Name
- an IceBox.ServiceManager.Identity property for an IceBox server
- an IceBox.LoadOrder property for an IceBox server, listing the IceBox services in the order they appear in the <server> descriptor

These files, located in the IcePack node's data directory, generally should not be edited by hand, because any changes will be lost if the application is redeployed.

21.8.6 Deployment Example

To illustrate all this, let us deploy a very simple application. The application descriptor is contained in the file application.xml, shown below:

```
<application>
    <node name="node" basedir="${basedir}">
        <server name="SimpleServer" binpath=".//server"
               descriptor="server.xml"/>
    </node>
</application>
```

Here's the server descriptor in the file server.xml:

```
<server kind="cpp">
    <adapters>
        <adapter name="SimpleAdapter" endpoints="default">
            <object identity="${name}" type="::Simple"/>
        </adapter>
    </adapters>
</server>
```

We can deploy this application in a few simple steps.

1. Create a configuration file called config for the IcePack node. The file should contain the following properties:

```
# 
# The IcePack locator proxy.
#
```

```
Ice.Default.Locator=IcePack/Locator:default -p 12000

#
# IcePack registry configuration.
#
IcePack.Registry.Client.Endpoints=default -p 12000
IcePack.Registry.Server.Endpoints=default
IcePack.Registry.Internal.Endpoints=default
IcePack.Registry.Admin.Endpoints=default
IcePack.Registry.Data=db/registry

#
# IcePack node configuration.
#
IcePack.Node.Name=node
IcePack.Node.Endpoints=default
IcePack.Node.Data=db/node
IcePack.Node.CollateRegistry=1
```

2. Create the database directory structure for the IcePack node:

```
$ mkdir db
$ mkdir db/registry
$ mkdir db/node
```

3. Start the IcePack node. Note that the node will have a collocated IcePack registry service, because the `IcePack.Node.CollateRegistry` property is defined in the configuration file.

```
$ icepacknode --Ice.Config=config
```

4. In another window, start the administration tool and deploy the application. We assume that the application descriptor resides in the file `application.xml`. When starting `icepackadmin`, we use the same configuration file as for `icepacknode`. This is done merely for convenience; the administration tool requires only the `Ice.Default.Locator` property.

```
$ icepackadmin --Ice.Config=config
>>> application add "application.xml"
```

The application should now be deployed. Note that `application.xml` must be enclosed in quotes because `application` is a keyword in the `icepack-admin` command language.

You can verify the configuration using the following commands:

```
>>> server list
SimpleServer
```

```
>>> adapter list
IcePack.Node-node
IcePack.Registry.Internal
SimpleAdapter-SimpleServer
>>> object find ::Simple
SimpleServer -t @ SimpleAdapter-SimpleServer
```

Notice that we did not explicitly assign an identifier to the object adapter in our server descriptor, so the deployer combined the adapter name with the server name to create the identifier `SimpleAdapter-SimpleServer`.

21.8.7 Deploying with Targets

Deployment descriptors allow you to define elements called *targets* that are deployed only when explicitly requested. This is useful in a number of situations, but one of the most common is debugging. We illustrate the benefits of using targets with two examples.

Example 1

Let's modify the server descriptor of our simple deployment example in Section 21.8.6 to include a `target` element:

```
<server kind="cpp">
    <adapters>
        <adapter name="SimpleAdapter" endpoints="default">
            <object identity="${name}" type="::Simple"/>
        </adapter>
    </adapters>

    <target name="debug">
        <properties>
            <property name="${name}.Debug" value="1"/>
        </properties>
    </target>
</server>
```

We have added a target named `debug` containing a `properties` element which defines one property. If this descriptor is deployed without specifying a target, the contents of the `target` element are simply ignored. When deployed with the `debug` target, all of the other non-target elements are processed as usual, and in addition, a property is added to the server's configuration file (see Section 21.8.5). The property name is composed using a descriptor variable, as described in

Section 21.9.6. In this example, the property name evaluates to `SimpleServer.Debug`.

Using `icepackadmin`, we can deploy this application normally as follows:

```
>>> application add "application.xml"
```

Alternatively, when we need to run the application with the debugging property defined, we can deploy it with the debugging target like this:

```
>>> application add "application.xml" debug
```

Notice that we can specify the target name as `debug`, even though the application descriptor has no target with that name (we added the target to the server descriptor). An unqualified name such as `debug` will be applied to every descriptor in the application. If there were multiple servers, and we wanted to ensure that debugging was enabled in only one of them, we would need to use a fully-qualified target name:

```
>>> application add "application.xml"
    "node.SimpleServer.debug"
```

Example 2

Now let us consider a slightly more complicated example. We have expanded the `target` element in our server descriptor:

```
<server kind="cpp">
    <adapters>
        <adapter name="SimpleAdapter" endpoints="default">
            <object identity="${name}" type="::Simple"/>
        </adapter>
    </adapters>

    <target name="debug">
        <adapters>
            <adapter name="DebugAdapter" endpoints="default">
                <object identity="DebugController"
                    type="::DebugController"/>
            </adapter>
        </adapters>
        <properties>
            <property name="${name}.Debug" value="1"/>
        </properties>
    </target>
</server>
```

In addition to the property introduced in the previous example, we have defined a new object adapter named `DebugAdapter` and registered a new object with the identity `DebugController`. The server creates these entities when it sees the `debug` property is set, providing clients with access to debugging facilities.

21.9 Descriptor Reference

This section describes each of the XML elements that comprise the IcePack deployment descriptors.

21.9.1 Application Elements

An application descriptor consists of a single `application` element containing one or more `node` elements. Each `node` element contains one or more `server` elements describing the servers to be deployed on that node.

Application

The `application` element supports the following attributes:

Table 21.1. Attributes for application element.

Attribute	Description	Required
<code>basedir</code>	The base directory from which all pathnames in this element are resolved. If not specified, the directory in which the descriptor file resides is used.	No

Node

The `node` element supports the following attributes:

Table 21.2. Attributes for node element.

Attribute	Description	Required
<code>name</code>	The name of the node as defined by the <code>Ice- Pack.Node.Name</code> configuration property.	Yes
<code>basedir</code>	The base directory from which all pathnames in this element are resolved. If not specified, the IcePack node's current working directory is used. The directory must be accessible to the IcePack node.	No

Server

The `server` element supports the following attributes:

Table 21.3. Attributes for server element.

Attribute	Description	Required
<code>name</code>	The name of the server. Server names uniquely identify servers in the IcePack registry.	Yes
<code>descriptor</code>	The pathname of the descriptor.	Yes
<code>binpath</code>	The pathname of the server executable. This attribute is required for C++ Ice servers. For C++ IceBox servers, the value <code>icebox</code> is used if not specified. For Java Ice and IceBox servers, the value <code>java</code> is used if not specified.	Only for C++ Ice servers
<code>libpath</code>	For Java Ice and IceBox servers, this attribute specifies the Java classpath. The value of this attribute is passed directly to the Java virtual machine. This attribute is not currently used for C++ servers.	No
<code>targets</code>	The names of targets used to deploy the server, separated by white space.	No

21.9.2 Server Elements

A server descriptor consists of a single `server` element. The element may contain the following:

- `service` elements defining IceBox services
- an `adapters` element defining object adapters to be registered in the IcePack registry
- a `properties` element specifying configuration properties for the server
- a `classname` element defining the name of the class containing the Java server's main method
- a `pwd` element specifying the pathname of the server's working directory
- `option` elements providing server command-line options
- `vm-option` elements providing command-line options for the Java Virtual Machine
- `env` elements providing environment variable definitions
- `target` elements defining deployment targets.

Server

The `server` element supports the following attributes:

Table 21.4. Attributes for server element.

Attribute	Description	Required
<code>basedir</code>	The base directory from which all pathnames in this element are resolved. If not specified, the directory in which the descriptor file resides is used.	No
<code>kind</code>	The type of server described by this descriptor. Possible values are: <code>cpp</code> , <code>java</code> , <code>cpp-icebox</code> and <code>java-icebox</code> .	Yes
<code>endpoints</code>	For C++ or Java IceBox servers, this attribute defines the endpoints for the IceBox service manager interface and is equivalent to the <code>IceBox.ServiceManager.Endpoints</code> configuration property.	Only for IceBox servers

Service

The `service` element is only valid in the scope of a `server` element if the server's kind attribute is `java-icebox` or `cpp-icebox`. The following attributes are supported:

Table 21.5. Attributes for service element.

Attribute	Description	Required
<code>name</code>	The name of the service. Service names uniquely identify services in an IceBox server.	Yes
<code>descriptor</code>	The pathname of the descriptor.	Yes
<code>targets</code>	The names of targets used to deploy the service, separated by white space.	No

Classname

The `classname` element is only valid in the scope of a `server` element if the server's kind attribute is `java`. The element value specifies the name of the class containing the server's `main` method. The element has no attributes.

Env

The `env` element defines an environment variable to be present in the server's environment upon activation. The element value should be of the form `name=value`. The element has no attributes.

Pwd

The `pwd` element value specifies the pathname of the server's working directory. The element has no attributes.

Option

The `option` element value defines a command-line option that is passed to the server upon activation. The element has no attributes.

Vm-option

The `vm-option` element is only valid in the scope of a `server` element if the server's kind attribute is `java` or `java-icebox`. The element value defines a

command-line option for the Java Virtual Machine, therefore it appears on the JVM command line prior to the server classname. The element has no attributes.

21.9.3 Service Elements

A service descriptor defines an IceBox service and consists of a single `service` element. The element may contain an `adapters` element defining object adapters to be registered in the IcePack registry, a `properties` element specifying configuration properties for the service, or `target` elements defining deployment targets.

Service

The `service` element supports the following attributes:

Table 21.6. Attributes for service element.

Attribute	Description	Required
<code>basedir</code>	The base directory from which all pathnames in this element are resolved. If not specified, the directory in which the descriptor file resides is used.	No
<code>kind</code>	The type of IceBox service defined by this descriptor. Possible values are <code>standard</code> or <code>freeze</code> .	Yes
<code>entry</code>	The service entry point. In C++, the entry point has the form <i>library:symbol</i> , where <i>library</i> is the name of a shared library or DLL, and <i>symbol</i> is the name of a factory function used to create the service. In Java, the entry point is the name of the service implementation class.	Yes
<code>dbenv</code>	If the value of the <code>kind</code> attribute is <code>freeze</code> , this attribute defines the service database environment directory. If not specified, the node provides a default database environment directory.	No

21.9.4 Common Elements

Deployment descriptors support a number of common elements.

Target

A `target` element encloses other elements that will be deployed only if the user specifies the target's name during deployment. A `target` element can be used in application, server, and service descriptors. Furthermore, a `target` element can only contain those elements that may legally appear in its enclosing descriptor. For example, a `node` element is legal inside a `target` element of an application descriptor, but not inside a `target` element of a server descriptor. The following attributes are supported:

Table 21.7. Attributes for target element.

Attribute	Description	Required
<code>name</code>	The target name.	Yes

Adapters

The `adapters` element contains one or more `adapter` elements and can be used in server and service descriptors. The element has no attributes.

Adapter

An `adapter` element instructs the deployer to register the adapter with the IcePack registry, which associates the adapter with its server and thereby enables

on-demand activation. The `adapter` element may only be used within an `adapters` element. The following attributes are supported:

Table 21.8. Attributes for adapter element.

Attribute	Description	Required
<code>name</code>	The object adapter name. This is the same name that an application passes to the <code>createObject-Adapter</code> operation.	Yes
<code>id</code>	The object adapter identifier. This identifier is used by the IcePack registry to uniquely identify an object adapter. If not specified, the identifier is created based on the adapter name and server name (and service name if the adapter is defined in a service descriptor).	No
<code>endpoints</code>	Endpoints for the object adapter.	Yes
<code>register</code>	If true, the <code>RegisterProcess</code> property is defined for this adapter. Only one adapter should be registered per server. See Section 21.6.3 for more information. If not defined, the default value is false.	No

Object

An `object` element instructs the deployer to register the object with the IcePack registry, to allow clients to lookup objects by identity or type. The `object` element may only be used in an `adapter` element. The following attributes are supported:

Table 21.9. Attributes for object element.

Attribute	Description	Required
<code>identity</code>	A stringified object identity.	Yes
<code>type</code>	The fully-scoped Slice type of the object, such as <code>::Hello</code> .	No

Properties

The `properties` element contains one or more `property` elements and can be used in server and service descriptors. The element has no attributes.

Property

Every deployed server or service has a configuration file located in the data directory of the IcePack node, and the `property` element instructs the deployer to add a property to this configuration file. The `property` element may only be used within a `properties` element. The following attributes are supported:

Table 21.10. Attributes for property element.

Attribute	Description	Required
<code>name</code>	The property name.	Yes
<code>value</code>	The property value.	If location is not specified
<code>location</code>	The property value interpreted as a pathname. A relative pathname is relative to the descriptor base directory.	If value is not specified

The `value` and `location` attributes are mutually exclusive.

21.9.5 Pathname Semantics

Relative pathnames in deployment descriptors are resolved using the base directory of the enclosing element. The default base directory is the directory in which the descriptor file resides, unless overridden by a `basedir` attribute as shown in the application descriptor in Section 21.8.1.

It is important to understand how these pathnames are used during deployment:

1. The `icepackadmin` tool (described in Section 21.7) sends the application descriptor pathname to the IcePack registry, which reads the file. The file must therefore be accessible to the registry, and relative pathnames will be resolved relative to the current working directory of the registry process.
2. The IcePack registry processes each `node` element by contacting the designated IcePack node to deploy its servers.
3. The IcePack node reads the server descriptor files, as well as any IceBox service descriptor files, therefore the pathnames of server and service descriptors must be accessible to the IcePack node on which they will be deployed,

and relative pathnames will be resolved relative to the current working directory of the node process. Furthermore, the pathnames for binaries and libraries must also be accessible to the IcePack node.

21.9.6 Descriptor Variables

A number of variables can be used in deployment descriptors. Variables have the form `${name}`, where `name` is the variable name. You can use variables to compose an attribute value, as shown below:

```
<object id="Factory-${name}-Object" type="" />
```

During deployment, `${name}` will be substituted with the descriptor name of the server or service.

These are the variables you can use in IcePack descriptors:

- `basedir`

The base directory of the descriptor. If not overridden by the top-level element `basedir` attribute, the base directory is the directory in which the descriptor resides. See Section 21.9.5 for more information on descriptor pathnames.

- `name`

The component name. In a server descriptor, this variable is set to the server name defined in the application descriptor. For an IceBox service, this variable is set to the service name defined in the IceBox server descriptor. This variable cannot be used in an application descriptor.

- `parent`

The name of the parent component. In a server descriptor, this variable is substituted with the name of the IcePack node where the server is deployed. In a service descriptor, this variable is substituted by the name of the IceBox server where the service is deployed. This variable cannot be used in an application descriptor.

- `fqn`

The fully qualified name of the component. In a server descriptor, this variable has the form `node_name.server_name`, where `node_name` is the name of the node where the server is deployed and `server_name` is the name of the server. In a service descriptor, this variable has the form `node_name.server_name.service_name`. This variable cannot be used in an application descriptor.

21.10 Troubleshooting

This section provides suggestions on how to resolve several common IcePack issues.

21.10.1 Activation Failure

Automatic server activation can fail for a number of reasons, but the most likely cause is incorrect configuration. For example, an IcePack node may fail to activate a server because the server's executable file or associated shared libraries could not be found. There are several steps you can take in this case:

1. Enable activation tracing in the IcePack node by setting the configuration property `IcePack.Node.Trace.Activation=3`.
2. Examine the tracing output and verify the server's executable pathname and working directory are correct.
3. If the executable pathname is correct, and it is a relative pathname, then it may not be correct relative to the IcePack node's current working directory (see Section 21.9.5). Either replace the relative pathname with an absolute pathname, or restart the IcePack node in the proper working directory.
4. Verify that the IcePack node is started with the correct PATH or `LD_LIBRARY_PATH` settings for the server's shared libraries.

Another cause of activation failure is a server fault during startup. After you have confirmed that the IcePack node is successfully spawning the server process using the steps above, you should then check for signs of a server fault (e.g., on UNIX, look for a `core` file in the IcePack node's current working directory). See Section 21.10.4 for more information on server failures.

21.10.2 Proxy Failure

A client may receive `Ice::NotRegisteredException` if binding fails for an indirect proxy (see Section 21.3.2). This exception indicates that the proxy's object identity or object adapter is not registered with the Ice locator facility. The following steps may help you discover the cause of the exception:

1. Use `icepackadmin` (see Section 21.7) to verify that the object identity or object adapter identifier is actually registered, and that it matches what is used by the proxy:

```
>>> adapter list
```

```
...  
->>> object find ::Hello  
...
```

2. If the problem persists, review your configuration to ensure that the locator proxy used by the client matches the IcePack registry's client endpoints, and that those endpoints are accessible to the client (i.e., are not blocked by a firewall).
3. Finally, enable locator tracing in the client by setting the configuration property `Ice.Trace.Locator=1`, then run the client again to see if any log messages are emitted which may indicate the problem.

21.10.3 Deployment Failure

Pathnames in descriptors, especially relative pathnames, are a common source of deployment problems. When writing descriptor files, you should be familiar with the deployment process explained in Section 21.9.5. In short, server and service descriptor files are read by the IcePack node on which they are deployed, therefore the pathnames of these descriptors must be specified such that the IcePack node can access them. In practice, this means the descriptor files for a node's servers and services either must be copied to the host on which the node is running, or must be accessible via a network file system. Furthermore, the use of relative pathnames requires knowledge of the IcePack node's current working directory and therefore should be used with care.

21.10.4 Server Failure

Diagnosing a server failure can be difficult, especially when servers are activated automatically on remote hosts. Here are a few suggestions:

1. If the server is running on a UNIX host, check the current working directory of the IcePack node process for signs of a server failure, such as a `core` file.
2. Judicious use of tracing can help to narrow the search. For example, if the failure occurs as a result of an operation invocation, enable protocol tracing in the Ice run time by setting the configuration property `Ice.Trace.Protocol=1` to discover the object identity and operation name of all requests.

Of course, the default log output channels (standard out and standard error) will probably be lost if the server is activated automatically, so either start the

server manually (see below) or redirect the log output (see Appendix C for a description of the `Ice.UseSyslog` property).

You can also use the `Ice::Logger` interface to emit your own trace messages.

3. Run the server in a debugger; a server configured for automatic activation can also be started manually if necessary. However, since the IcePack node did not activate the server, it cannot monitor the server process and therefore will not know when the server terminates. This will prevent subsequent activation unless you clean up the IcePack state when you have finished debugging and terminated the server. You can do this by starting the server using `icepack-admin` (see Section 21.7):

```
>>> server start TheServer
```

This will cause the IcePack node to activate (and therefore monitor) the server process. If you do not want to leave the server running, you can stop it with the equivalent `icepackadmin` command.

4. After the server is activated as is in a quiescent state, attach your debugger to the running server process. This avoids the issues associated with starting the server manually (as described in the previous step), but does not provide as much flexibility in customizing the server's startup environment.

21.11 Summary

This chapter provided a detailed discussion of IcePack, including the modifications required to incorporate IcePack into client and server applications, as well as the configuration and administration of IcePack components. Once you understand the basic concepts on which IcePack is founded, you quickly begin to appreciate the flexibility, power, and convenience of IcePack's capabilities:

- automatic server activation increases reliability and makes more efficient use of processing resources
- the locator facility simplifies administrative requirements and minimizes coupling between clients and servers
- the deployment mechanism reduces mundane configuration tasks into a set of easily-understood, reusable XML files

In short, IcePack provides the tools you need to develop robust, enterprise-class Ice applications.

Chapter 22

Freeze

22.1 Chapter Overview

This chapter describes how to use Freeze to add persistence to Ice applications. Section 22.3 discusses the Freeze map and shows how to use it in a simple example. Section 22.4 examines an implementation of the file system using a Freeze map. Section 22.5 presents the Freeze evictor, and Section 22.6 demonstrates the Freeze evictor in another file system implementation.

22.2 Introduction

Freeze represents a set of persistence services, as shown in Figure 22.1.

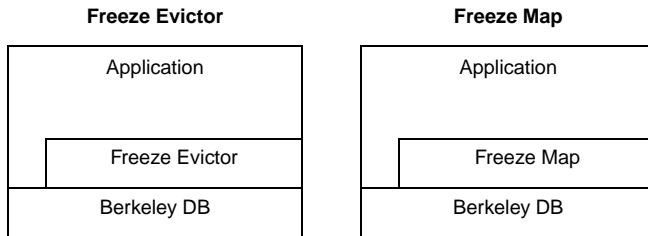


Figure 22.1. Layer diagram for Freeze persistence services.

The Freeze persistence services are described below:

- The Freeze evictor is a highly-scalable implementation of an Ice servant locator that provides automatic persistence and eviction of Ice objects with only minimal application code.
- The Freeze map is a generic associative container. Code generators are provided which produce type-specific maps for any Slice key and value types. Applications interact with a Freeze map just like any other associative container, except the keys and values of a Freeze map are persistent.

As you will see from the examples in this chapter, integrating a Freeze map or evictor into your Ice application is quite straightforward: once you define your persistent data in Slice, Freeze manages the mundane details of persistence.

Freeze is implemented using Berkeley DB, a compact and high-performance embedded database. The Freeze map and evictor APIs insulate applications from the Berkeley DB API, but do not prevent applications from interacting directly with Berkeley DB if necessary.

22.3 The Freeze Map

A Freeze map is a persistent, associative container in which the key and value types can be any primitive or user-defined Slice types. For each pair of key and value types, the developer uses a code-generation tool to produce a language-specific class that conforms to the standard conventions for maps in that language.

For example, in C++ the generated class resembles an STL `map`, and in Java it implements the `java.util.Map` interface. Most of the logic for storing and retrieving state to and from the database is implemented in a Freeze base class. The generated map classes derive from this base class, so they contain little code and therefore are efficient in terms of code size.

You can only store data types that are defined in Slice in a Freeze map. Types without a Slice definition (that is, arbitrary C++ or Java types) cannot be stored because a Freeze map reuses the Ice-generated marshalling code to create the persistent representation of the data in the database. This is especially important to remember when defining a Slice class whose instances will be stored in a Freeze map; only the “public” (Slice-defined) data members will be stored, not the private state members of any derived implementation class.

22.3.1 Freeze Connections

In order to create a Freeze Map object, you first need to obtain a Freeze Connection object by connecting to a database environment.

As illustrated in Figure 22.2, a Freeze Map is associated with a single connection and a single database file. Connection and map objects are “single threaded”: if you want to use a connection or any of its associated maps from multiple threads, you must serialize access to them. If your application requires concurrent access to the same database file (persistent Map), you must create several connections and associated maps.

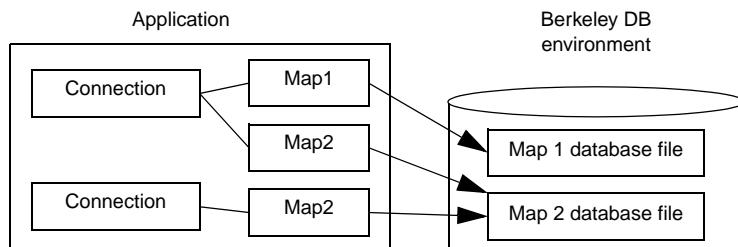


Figure 22.2. Freeze Connections and Maps.

22.3.2 Transactions

You may optionally use transactions with Freeze maps. Freeze transactions provide the usual ACID (atomicity, concurrency, isolation, durability) properties. For example, a transaction allows you to group several database updates in one atomic unit: either all or none of the updates within the transaction occur.

A transaction is started using the `beginTransaction` operation on the `Connection` object. Once a connection has an associated transaction, all operations on the map objects associated with this connection use this transaction. Eventually, you end the transaction by calling `commit` or `rollback`: `commit` saves all your updates while `rollback` undoes them.

```
module Freeze {  
  
    local interface Transaction {  
        void commit();  
        void rollback();  
    };  
  
    local interface Connection {  
        Transaction beginTransaction();  
        nonmutating Transaction currentTransaction();  
        // ...  
    };  
};
```

If you do not use transactions, every non-iterator update is enclosed in its own internal transaction, and every read-write iterator has an associated internal transaction which is committed when the iterator is closed.

22.3.3 Iterators

Iterators allow you to traverse the contents of a Freeze map. Iterators are implemented using Berkeley DB cursors and acquire locks on the underlying database page files. In C++, both read-only (`const_iterator`) and read-write iterators (`iterator`) are available; in Java only read-write iterators are supported.

Locks held by an iterator are released when the iterator is closed (if you do not use transactions) or when the enclosing transaction is ended. Releasing locks held by iterators is very important to let other threads access the database file through other connection and map objects. Occasionally it is even necessary to release locks to avoid self-deadlock (waiting forever for a lock held by an iterator created by the same thread).

To improve ease of use and make self-deadlocks less likely, Freeze often closes iterators automatically. When you start or when you end a transaction, Freeze closes all the iterators associated with the corresponding maps. If you do not use transactions, any write operation on a map (such as inserting a new element) automatically closes all iterators opened on the same map object, except for the current iterator when the write operation is performed through that iterator. In Java, iterators are also closed when the enclosing map or connection is closed by the application.

There is, however, one situation where an explicit iterator close is needed to avoid self-deadlock:

- you do not use transactions, and
- you have an opened iterator that was used to update a map (it holds a write lock), and
- in the same thread, you read that map.

Read operations never close iterators automatically. In that situation, you need to either use transactions or explicitly close the iterator that holds the write lock.

22.3.4 Recovering from Deadlock Exceptions

If you use multiple threads to access a database file, Berkeley DB may acquire locks in conflicting orders (on behalf of different transactions or iterators). For example, an iterator could have a read-lock on page P1 and attempt to acquire a write-lock on page P2, while another iterator (on a different map object associated with the same database file) could have a read-lock on P2 and attempt to acquire a write-lock on P1.

When this occurs, Berkeley DB detects a deadlock and resolves it by returning a “deadlock” error to one or more threads. For all non-iterator operations performed outside any transaction, such as an insertion into a map, Freeze catches such errors and automatically retries the operation until it succeeds. For other operations, Freeze reports this deadlock by raising `Freeze::DeadlockException`. In that case, the associated transaction or iterator is also automatically rolled back or closed. A properly written application is expected to catch deadlock exceptions and retry the transaction or iteration.

22.3.5 Using a Simple Map in C++

As an example, the following command generates a simple map:

```
$ slice2freeze --dict StringIntMap,string,int StringIntMap
```

The **slice2freeze** compiler creates C++ classes for Freeze maps. The command above directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. The final argument is the base name for the output files, to which the compiler appends the `.h` and `.cpp` suffixes. Therefore, this command produces two C++ source files: `StringIntMap.h` and `StringIntMap.cpp`.

Here is a simple program that demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit `read` or `write` operations called by the program; instead, simply using the map has the side effect of accessing the database.

```
#include <Freeze/Freeze.h>
#include <StringIntMap.h>

int
main(int argc, char* argv[])
{
    // Initialize the Communicator.
    //
    Ice::CommunicatorPtr communicator =
        Ice::initialize(argc, argv);

    // Create a Freeze database connection.
    //
    Freeze::ConnectionPtr connection =
        Freeze::createConnection(communicator, "db");

    // Instantiate the map.
    //
    StringIntMap map(connection, "simple");

    // Clear the map.
    //
    map.clear();

    Ice::Int i;
    StringIntMap::iterator p;

    // Populate the map.
    //
    for (i = 0; i < 26; i++) {
        std::string key(1, 'a' + i);
        map.insert(make_pair(key, i));
    }
}
```

```
    }

    // Iterate over the map and change the values.
    //
    for (p = map.begin(); p != map.end(); ++p)
        p.set(p->second + 1);

    // Find and erase the last element.
    //
    p = map.find("z");
    assert(p != map.end());
    map.erase(p);

    // Clean up.
    //
    connection->close();
    communicator->destroy();

    return 0;
}
```

Prior to instantiating a Freeze map, the application must connect to a Berkeley DB database environment.

```
Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator, "db");
```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, which by default is created if it does not exist. Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.

```
StringIntMap map(connection, "simple");
```

Next, we clear the map. This ensures we have an empty map (and therefore an empty database) in case the program is run more than once.

```
map.clear();
```

We populate the map, using a single-character string as the key. The Freeze map supports several `insert` methods for adding entries, similar to an STL map. Insertion via operator`[]` is not supported.

```
for (i = 0; i < 26; i++) {
    std::string key(1, 'a' + i);
    map.insert(make_pair(key, i));
}
```

Iterating over the map will look familiar to STL users. However, to modify a value at the iterator's current position, you must use the nonstandard `set` method:

```
for (p = map.begin(); p != map.end(); ++p)
    p.set(p->second + 1);
```

Next, the program obtains an iterator positioned at the element with key `z`, and erases it.

```
p = map.find("z");
assert(p != map.end());
map.erase(p);
```

Finally, the program closes the database connection, destroys its communicator and terminates.

```
connection->close();
```

It is not necessary to explicitly close the database connection, but we demonstrate it here for the sake of completeness.

22.3.6 Using a Simple Map in Java

As an example, the following command generates a simple map:

```
$ slice2freezej --dict StringIntMap,string,int
```

The `slice2freezej` compiler creates Java classes for Freeze maps. The command above directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. This command produces one Java source file: `StringIntMap.java`.

Here is a simple program that demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit `read` or `write` operations called by the program; instead, simply using the map has the side effect of accessing the database.

```
public class Client
{
    public static void
    main(String[] args)
    {
        // Initialize the Communicator.
```

```
//  
Ice.Communicator communicator = Ice.Util.initialize(args);  
  
// Create a Freeze database connection.  
//  
Freeze.Connection connection =  
    Freeze.Util.createConnection(communicator, "db");  
  
// Instantiate the map.  
//  
StringIntMap map =  
    new StringIntMap(connection, "simple", true);  
  
// Clear the map.  
//  
map.clear();  
  
int i;  
java.util.Iterator p;  
  
// Populate the map.  
//  
for (i = 0; i < 26; i++) {  
    final char[] ch = { (char)('a' + i) };  
    map.put(new String(ch), new Integer(i));  
}  
  
// Iterate over the map and change the values.  
//  
p = map.entrySet().iterator();  
while (p.hasNext()) {  
    java.util.Map.Entry e = (java.util.Map.Entry)p.next();  
    Integer in = (Integer)e.getValue();  
    e.setValue(new Integer(in.intValue() + 1));  
}  
  
// Find and erase the last element.  
//  
boolean b;  
b = map.containsKey("z");  
assert(b);  
b = map.fastRemove("z");  
assert(b);  
  
// Clean up.  
//
```

```
        map.close();
        connection.close();
        communicator.destroy();

        System.exit(0);
    }
}
```

Prior to instantiating a Freeze map, the application must connect to the Berkeley DB database environment.

```
Freeze.Connection connection =
    Freeze.Util.createConnection(communicator, "db");
```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, and the third argument is a flag indicating whether the database should be created if it does not exist. Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.

```
StringIntMap map = new StringIntMap(connection, "simple", true);
```

Next, we clear the map. This ensures we have an empty map (and therefore an empty database) in case the program is run more than once.

```
map.clear();
```

We populate the map, using a single-character string as the key. As with `java.util.Map`, the key and value types must be Java objects.

```
for (i = 0; i < 26; i++) {
    final char[] ch = { (char)('a' + i) };
    map.put(new String(ch), new Integer(i));
}
```

Iterating over the map is no different from iterating over any other map that implements the `java.util.Map` interface:

```
p = map.entrySet().iterator();
while (p.hasNext()) {
    java.util.Map.Entry e =
        (java.util.Map.Entry)p.next();
    Integer in = (Integer)e.getValue();
    e.setValue(new Integer(in.intValue() + 1));
}
```

Next, the program verifies that an element exists with key z, and then removes it. Note that the program uses a non-standard method for removing the element. The `fastRemove` method differs from the standard `remove` method in that it does not return the value associated with the removed element; instead it returns a boolean indicating whether the element was found. By eliminating the need to return the element value, the method avoids the overhead of reading the value from the database and decoding it. The standard `remove` method could also be used here; we chose to use `fastRemove` instead simply to demonstrate its use:

```
b = map.containsKey("z");
assert(b);
b = map.fastRemove("z");
assert(b);
```

Finally, the program closes the map and its connection.

```
map.close();
connection.close();
```

22.4 Using a Freeze Map in the File System Server

We can use a Freeze map to add persistence to the file system server, and we present C++ and Java implementations in this section. However, as you will see in Section 22.5, a Freeze evictor is often a better choice for applications (such as the file system server) in which the persistent value is an Ice object.

In general, incorporating a Freeze map into your application requires the following steps:

1. Evaluate your existing Slice definitions for suitable key and value types.
2. If no suitable key or value types are found, define new (possibly derived) types that capture your persistent state requirements. Consider placing these definitions in a separate file: these types are only used by the server for persistence, and therefore do not need to appear in the “public” definitions required by

clients. Also consider placing your persistent types in a separate module to avoid name clashes.

3. Generate a Freeze map for your persistent types using the Freeze compiler.
4. Use the Freeze map in your operation implementations.

22.4.1 Choosing Key and Value Types

Our goal is to implement the file system using a Freeze map for all persistent storage, including files and their contents. Our first step is to select the Slice types we will use for the key and value types of our map. We will keep the same basic design as in XREF, therefore we need a suitable representation for persistent files and directories, as well as a unique identifier for use as a key.

Conveniently enough, Ice objects already have a unique identifier of type `Ice::Identity`, and this will do fine as the key type for our map.

Unfortunately, the selection of a value type is more complicated. Looking over the `Filesystem` module in XREF, we do not find any types that capture all of our persistent state, so we need to extend the module with some new types:

```
module Filesystem {  
    class PersistentNode {  
        string name;  
    };  
  
    class PersistentFile extends PersistentNode {  
        Lines text;  
    };  
  
    class PersistentDirectory extends PersistentNode {  
        NodeDict nodes;  
    };  
};
```

Our Freeze map will therefore map from `Ice::Identity` to `PersistentNode`, where the values are actually instances of the derived classes `PersistentFile` or `PersistentDirectory`. If we had followed the advice at the beginning of Section 22.4, we would have defined `File` and `Directory` classes in a separate `PersistentFilesystem` module, but in this example we use the existing `Filesystem` module for the sake of simplicity.

22.4.2 Implementing the File System Server in C++

In this section we present a C++ file system implementation that utilizes a Freeze map for persistent storage. The implementation is based on the one discussed in XREF, and, in this section, we only discuss code that illustrates use of the Freeze map.

Generating the Map

Now that we have selected our key and value types, we can generate the map as follows:

```
$ slice2freeze -I$(ICE_HOME)/slice --dict \
    IdentityNodeMap,Ice::Identity,Filesystem::PersistentNode \
    IdentityNodeMap Filesystem.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
```

The resulting map class is named `IdentityNodeMap`.

The Server main Program

The server's main program is responsible for initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice::Application`, as described in Section 10.3.1. Our server main program has now become the following:

```
#include <FilesystemI.h>
#include <Ice/Application.h>
#include <Freeze/Freeze.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : public virtual Ice::Application {
public:
    FilesystemApp(const string & envName) :
        _envName(envName) { }

    virtual int run(int, char * []) {
        // Terminate cleanly on receipt of a signal
        //
        shutdownOnInterrupt();

        // Install object factories
        //
        communicator()->addObjectFactory(
```

```
PersistentFile::ice_factory(),
PersistentFile::ice_staticId());

communicator()->addObjectFactory(
    PersistentDirectory::ice_factory(),
    PersistentDirectory::ice_staticId());

// Create an object adapter (stored in the NodeI::_adapter
// static member)
//
NodeI::_adapter = communicator()->
    createObjectAdapterWithEndpoints(
        "FreezeFilesystem", "default -p 10000");

//
// Set static members used to create connections and maps
//
NodeI::_communicator = communicator();
NodeI::_envName = _envName;
NodeI::_dbName = "mapfs";

// Find the persistent node for the root directory, or
// create it if not found
//
Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator(), _envName);
IdentityNodeMap persistentMap(connection, NodeI::_dbName);

Ice::Identity rootId = Ice::stringToIdentity("RootDir");
PersistentDirectoryPtr pRoot;
{
    IdentityNodeMap::iterator p =
        persistentMap.find(rootId);

    if (p != persistentMap.end()) {
        pRoot =
            PersistentDirectoryPtr::dynamicCast(
                p->second);
        assert(pRoot);
    } else {
        pRoot = new PersistentDirectory;
        pRoot->name = "/";
        persistentMap.insert(
            IdentityNodeMap::value_type(rootId, pRoot));
    }
}
```

```
// Create the root directory (with name "/" and no parent)
//
DirectoryIPtr root = new DirectoryI(rootId, pRoot, 0);

// Ready to accept requests now
//
NodeI::_adapter->activate();

// Wait until we are done
//
communicator()->waitForShutdown();
if (interrupted()) {
    cerr << appName()
        << " : received signal, shutting down" << endl;
}

return 0;
}

private:
    string _envName;

};

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}
```

Let us examine the changes in detail. First, we are now including `Freeze/Freeze.h` instead of `Ice/Ice.h`. This `Freeze` header file includes all of the other `Freeze` (and `Ice`) header files this source file requires.

Next, we define the class `FilesystemApp` as a subclass of `Ice::Application`, and provide a constructor taking a string argument:

```
FilesystemApp(const string & envName) :
    _envName(envName) { }
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

One of the first tasks `run` performs is installing the `Ice` object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not

exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. Since these Slice classes have no operations, we can use their built-in factories.

```
communicator()->addObjectFactory(
    PersistentFile::ice_factory(),
    PersistentFile::ice_staticId());

communicator()->addObjectFactory(
    PersistentDirectory::ice_factory(),
    PersistentDirectory::ice_staticId());
```

Next, we set all the `NodeI` static members.

```
NodeI::_adapter = communicator()->
    createObjectAdapterWithEndpoints(
        "FreezeFilesystem", "default -p 10000");

NodeI::_communicator = communicator();
NodeI::_envName = _envName;
NodeI::_dbName = "mapfs";
```

Then we create a Freeze connection and a Freeze map. When the last connection to a Berkeley DB environment is closed, Freeze automatically closes this environment, so keeping a connection in the main function ensures the underlying Berkeley DB environment remains open. Likewise, we keep a map in the `main` function to keep the underlying Berkeley DB database open.

```
Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator(), _envName);
IdentityNodeMap persistentMap(connection, NodeI::_dbName);
```

Now we need to initialize the root directory node. We first query the map for the identity of the root directory node; if no match is found, we create a new `PersistentDirectory` instance and insert it into the map. We use a scope to close the iterator after use; otherwise, this iterator could keep locks and prevent subsequent access to the map through another connection.

```
Ice::Identity rootId = Ice::stringToIdentity("RootDir");
PersistentDirectoryPtr pRoot;
{
    IdentityNodeMap::iterator p =
        persistentMap.find(rootId);
```

```

        if (p != persistentMap.end()) {
            pRoot =
                PersistentDirectoryPtr::dynamicCast(
                    p->second);
            assert(pRoot);
        } else {
            pRoot = new PersistentDirectory;
            pRoot->name = "/";
            persistentMap.insert(
                IdentityNodeMap::value_type(rootId, pRoot));
        }
    }
}

```

Finally, the `main` function instantiates the `FilesystemApp`, passing `db` as the name of the database environment.

```

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}

```

The Servant Class Definitions

We also must change the servant classes to incorporate the Freeze map. We are maintaining the multiple-inheritance design from XREF, but we have added some methods and changed the constructor arguments and state members.

Let us examine the definition of `NodeI` first. You will notice the addition of the `getPersistentNode` method, which allows `NodeI` to gain access to the persistent node in order to implement the `Node` operations. Another alternative would have been to add a `PersistentNodePtr` member to `NodeI`, but that would have forced the `FileI` and `DirectoryI` classes to downcast this member to the appropriate subclass.

```

namespace Filesystem {
    class NodeI : virtual public Node {
    public:
        // ... Ice operations ...
        static Ice::ObjectAdapterPtr _adapter;
        static Ice::CommunicatorPtr _communicator;
        static std::string _envName;
        static std::string _dbName;
    protected:
        NodeI(const Ice::Identity &, const DirectoryIPtr &);

        virtual PersistentNodePtr getPersistentNode() const = 0;
    };
}

```

```
PersistentNodePtr find(const Ice::Identity &) const;
IdentityNodeMap _map;
DirectoryIPtr _parent;
IceUtil::RecMutex _nodeMutex;
bool _destroyed;
public:
    const Ice::Identity _ID;
};
}
```

Other changes of interest in `NodeI` are the addition of the members `_map` and `_destroyed`, and we have changed the `NodeI` constructor to accept an `Ice::Identity` argument.

The `FileI` class now has a single state member of type `PersistentFilePtr`, representing the persistent state of this file. Its constructor has also changed to accept `Ice::Identity` and `PersistentFilePtr`.

```
namespace Filesystem {
    class FileI : virtual public File,
                  virtual public NodeI {
    public:
        // ... Ice operations ...
        FileI(const Ice::Identity &, const PersistentFilePtr &,
               const DirectoryIPtr &);

    protected:
        virtual PersistentNodePtr getPersistentNode() const;
    private:
        PersistentFilePtr _file;
    };
}
```

The `DirectoryI` class has undergone a similar transformation.

```
namespace Filesystem {
    class DirectoryI : virtual public Directory,
                      virtual public NodeI {
    public:
        // ... Ice operations ...
        DirectoryI(const Ice::Identity &,
                   const PersistentDirectoryPtr&,
                   const DirectoryIPtr &);

    protected:
        virtual PersistentNodePtr getPersistentNode() const;
        // ...
}
```

```
private:  
    // ...  
    PersistentDirectoryPtr _dir;  
};  
}
```

Implementing FileI

Let us examine how the implementations have changed. The `FileI` methods are still fairly trivial, but there are a few aspects that need discussion.

First, each operation now checks the `_destroyed` member and raises `Ice::ObjectNotExistException` if the member is `true`. This is necessary in order to ensure that the Freeze map is kept in a consistent state. For example, if we allowed the `write` operation to proceed after the file node had been destroyed, then we would have mistakenly added an entry back into the Freeze map for that file. Previous file system implementations ignored this issue because it was relatively harmless, but that is no longer true in this version.

Next, notice that the `write` operation calls `put` on the map after changing the `text` member of its `PersistentFile` object. Although the file's `PersistentFilePtr` member points to a value in the Freeze map, changing that value has no effect on the persistent state of the map until the value is reinserted into the map, thus overwriting the previous value.

Finally, the constructor now accepts an `Ice::Identity`. This differs from previous implementations in that the identity used to be created by the `NodeI` constructor. However, as we will see later, the caller needs to determine the identity prior to invoking the subclass constructors. Similarly, the constructor is not responsible for creating a `PersistentFile` object, but rather is given one. This accommodates our two use cases: creating a new file, and restoring an existing file from the map.

```
Filesystem::Lines  
Filesystem::FileI::read(const Ice::Current &) const  
{  
    IceUtil::RWRecMutex::RLock lock(_nodeMutex);  
  
    if (_destroyed)  
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);  
  
    return _file->text;  
}  
  
void  
Filesystem::FileI::write(const Filesystem::Lines & text,
```

```
                const Ice::Current &)
{
    IceUtil::RWRecMutex::WLock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    _file->text = text;
    _map.put(IdentityNodeMap::value_type(_ID, _file));
}

Filesystem::FileI::FileI(const Ice::Identity & id,
                        const PersistentFilePtr & file,
                        const DirectoryIPtr & parent) :
    NodeI(id, parent), _file(file)
{
}

Filesystem::PersistentNodePtr
Filesystem::FileI::getPersistentNode() const
{
    return _file;
}
```

Implementing DirectoryI

The `DirectoryI` implementation requires more substantial changes. We begin our discussion with the `createDirectory` operation.

```
Filesystem::DirectoryI::createDirectory(
    const std::string & name,
    const Ice::Current & current)
{
    IceUtil::RWRecMutex::WLock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    checkName(name);

    PersistentDirectoryPtr persistentDir
        = new PersistentDirectory;
    persistentDir->name = name;
    DirectoryIPtr dir = new DirectoryI(
        Ice::stringToIdentity(IceUtil::generateUUID()),
        persistentDir, this);
```

```

        assert(find(dir->_ID) == 0);
        _map.put(make_pair(dir->_ID, persistentDir));

        DirectoryPrx proxy = DirectoryPrx::uncheckedCast(
            current.adapter->createProxy(dir->_ID));

        NodeDesc nd;
        nd.name = name;
        nd.type = DirType;
        nd.proxy = proxy;
        _dir->nodes[name] = nd;
        _map.put(IdentityNodeMap::value_type(_ID, _dir));

        return proxy;
    }
}

```

After validating the node name, the operation creates a `PersistentDirectory`¹ object for the child directory, which is passed to the `DirectoryI` constructor along with a unique identity. Next, we store the child's `PersistentDirectory` object in the Freeze map. Finally, we initialize a new `NodeDesc` value and insert it into the parent's node table and then reinsert the parent's `PersistentDirectory` object into the Freeze map.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```

Filesystem::FilePrx
Filesystem::DirectoryI::createFile(const std::string & name,
                                    const Ice::Current & current)
{
    IceUtil::RWRecMutex::WLock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    checkName(name);

    PersistentFilePtr persistentFile = new PersistentFile;
    persistentFile->name = name;
    FileIPtr file = new FileI(
        Ice::stringToIdentity(IceUtil::generateUUID()),

```

1. Since this Slice class has no operations, the compiler generates a concrete class that an application can instantiate.

```

        persistentFile, this);
assert(find(file->_ID) == 0);
_map.put(make_pair(file->_ID, persistentFile));

FilePrx proxy = FilePrx::uncheckedCast(
    current.adapter->createProxy(file->_ID));

NodeDesc nd;
nd.name = name;
nd.type = FileType;
nd.proxy = proxy;
_dir->nodes[name] = nd;
_map.put(IdentityNodeMap::value_type(_ID, _dir));

return proxy;
}

```

The next significant change is in the `DirectoryI` constructor. The body of the constructor now instantiates all of its immediate children, which effectively causes all nodes to be instantiated recursively.

For each entry in the directory's node table, the constructor locates the matching entry in the `Freeze` map. The key type of the `Freeze` map is `Ice::Identity`, so the constructor obtains the key by invoking `ice_getIdentity` on the child's proxy.

```

Filesystem::DirectoryI::DirectoryI(
    const Ice::Identity & id,
    const PersistentDirectoryPtr & dir,
    const DirectoryIPtr & parent) :
    NodeI(id, parent), _dir(dir)
{
    // Instantiate the child nodes
    //
    for (NodeDict::iterator p = dir->nodes.begin();
         p != dir->nodes.end(); ++p) {
        Ice::Identity id = p->second.proxy->ice_getIdentity();
        PersistentNodePtr node = find(id);
        assert(node != 0);
        if (p->second.type == DirType) {
            PersistentDirectoryPtr pDir =
                PersistentDirectoryPtr::dynamicCast(node);
            assert(pDir);
            DirectoryIPtr d = new DirectoryI(id, pDir, this);
        } else {
            PersistentFilePtr pFile =

```

```
        PersistentFilePtr::dynamicCast(node);
        assert(pFile);
        FileIPtr f = new FileI(id, pFile, this);
    }
}
}
```

If it seems inefficient for the `DirectoryI` constructor to immediately instantiate all of its children, you are right. This clearly will not scale well to large node trees, so why are we doing it?

Previous implementations of the file system service returned transient proxies from `createDirectory` and `createFile`. In other words, if the server was stopped and restarted, any existing child proxies returned by the old instance of the server would no longer work. However, now that we have a persistent store, we should endeavor to ensure that proxies will remain valid across server restarts. There are a couple of implementation techniques that satisfy this requirement:

1. Instantiate all of the servants in advance, as shown in the `DirectoryI` constructor.
2. Use a servant locator.

We chose not to include a servant locator in this example because it complicates the implementation and, as we will see in Section 22.5, a Freeze evictor is ideally suited for this application and a better choice than writing a servant locator.

The last `DirectoryI` method we discuss is `removeChild`, which removes the entry from the node table, and then reinserts the `PersistentDirectory` object into the map to make the change persistent.

```
void
Filesystem::DirectoryI::removeChild(const string & name)
{
    IceUtil::RecMutex::Lock lock(_nodeMutex);

    _dir->nodes.erase(name);
    _map.put(IdentityNodeMap::value_type(_ID, _dir));
}
```

Implementing `NodeI`

There are a few changes to the `NodeI` implementation that should be mentioned. First, you will notice the use of `getPersistentNode` in order to obtain the node's name. We could have simply invoked the `name` operation, but that would incur the overhead of another mutex lock.

Then, the `destroy` operation removes the node from the `Freeze` map and sets the `_destroyed` member to true.

The `NodeI` constructor no longer computes a value for the identity. This is necessary in order to make our servants truly persistent. Specifically, the identity for a node is computed once when that node is created, and must remain the same for the lifetime of the node. The `NodeI` constructor therefore must not compute a new identity, but rather simply remember the identity that is given to it. The `NodeI` constructor also constructs the map object (`_map` data member). Remember this object is single-threaded: we use it only in constructors or when we have a write lock on the recursive read-write `_nodeMutex`.

Finally, the `find` function illustrates what needs to be done when iterating over a database that multiple threads can use concurrently. `find` catches `Freeze::DeadlockException` and retries.

```
std::string
Filesystem::NodeI::name(const Ice::Current &) const
{
    IceUtil::RecMutex::Lock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    return getPersistentNode()->name;
}

void
Filesystem::NodeI::destroy(const Ice::Current & current)
{
    IceUtil::RWRecMutex::WLock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    if (!_parent) {
        Filesystem::PermissionDenied e;
        e.reason = "cannot remove root directory";
        throw e;
    }

    _parent->removeChild(getPersistentNode()->name);
    _map.erase(current.id);
    current.adapter->remove(current.id);
    _destroyed = true;
}
```

```
Filesystem::NodeI::NodeI(const Ice::Identity & id,
                        const DirectoryIPtr & parent)
: _map(Freeze::createConnection(_communicator, _envName),
      _dbName),
  _parent(parent), _destroyed(false), _ID(id)
{
    // Add the identity of self to the object adapter
    //
    _adapter->add(this, _ID);
}

Filesystem::PersistentNodePtr
Filesystem::NodeI::find(const Ice::Identity & id) const
{
    for(;;) {
        try {
            IdentityNodeMap::const_iterator p = _map.find(id);
            if(p == _map.end())
                return 0;
            else
                return p->second;
        }
        catch(const Freeze::DeadlockException &) {
            // Try again
            //
        }
    }
}
```

22.4.3 Implementing the File System Server in Java

In this section we present a Java file system implementation that utilizes a Freeze map for persistent storage. The implementation is based on the one discussed in XREF; in this section we only discuss code that illustrates use of the Freeze map.

Generating the Map

Now that we have selected our key and value types, we can generate the map as follows:

```
$ slice2freezej -I$(ICE_HOME)/slice --dict \
  Filesystem.IdentityNodeMap,Ice::Identity, \
  Filesystem::PersistentNode \
  Filesystem.ice $(ICE_HOME)/slice/Ice/Identity.ice
```

The resulting map class is named `IdentityNodeMap` and is defined in the package `Filesystem`².

The Server main Program

The server's main program is responsible for initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice.Application` as described in Section 12.3.1. Our server main program has now become the following:

```
import Filesystem.*;

public class Server extends Ice.Application {
    public
    Server(String envName)
    {
        _envName = envName;
    }

    public int
    run(String[] args)
    {
        // Install object factories
        //
        communicator().addObjectFactory(
            PersistentFile.ice_factory(),
            PersistentFile.ice_staticId());
        communicator().addObjectFactory(
            PersistentDirectory.ice_factory(),
            PersistentDirectory.ice_staticId());

        // Create an object adapter (stored in the _adapter
        // static member)
        //
        Ice.ObjectAdapter adapter =

```

2. We cannot generate `IdentityNodeMap` in the unnamed (top-level) package because Java's name resolution rules would prevent the implementation classes in the `Filesystem` package from using it. See the Java Language Specification for more information.

```
communicator().createObjectAdapterWithEndpoints(
    "FreezeFilesystem", "default -p 10000");
DirectoryI._adapter = adapter;
FileI._adapter = adapter;

//
// Set static members used to create connections and maps
//
String dbName = "mapfs";
DirectoryI._communicator = communicator();
DirectoryI._envName = _envName;
DirectoryI._dbName = dbName;
FileI._communicator = communicator();
FileI._envName = _envName;
FileI._dbName = dbName;

// Find the persistent node for the root directory. If
// it doesn't exist, then create it.
Freeze.Connection connection =
    Freeze.Util.createConnection(
        communicator(), _envName);
IdentityNodeMap persistentMap =
    new IdentityNodeMap(connection, dbName, true);

Ice.Identity rootId =
    Ice.Util.stringToIdentity("RootDir");
PersistentDirectory pRoot =
    (PersistentDirectory)persistentMap.get(rootId);
if(pRoot == null)
{
    pRoot = new PersistentDirectory();
    pRoot.name = "/";
    pRoot.nodes = new java.util.HashMap();
    persistentMap.put(rootId, pRoot);
}

// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI(rootId, pRoot, null);

// Ready to accept requests now
//
adapter.activate();

// Wait until we are done
//
```

```
communicator().waitForShutdown();

// Clean up
//
connection.close();

return 0;
}

public static void
main(String[] args)
{
    Server app = new Server("db");
    app.main("Server", args);
    System.exit(0);
}

private String _envName;
}
```

Let us examine the changes in detail. First, we define the class `Server` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

```
public
Server(String envName)
{
    _envName = envName;
}
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

One of the first tasks `run` performs is installing the Ice object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. Since these Slice classes have no operations, we can use their built-in factories.

```
communicator(). addObjectFactory(
    PersistentFile.ice_factory(),
    PersistentFile.ice_staticId());
communicator(). addObjectFactory(
    PersistentDirectory.ice_factory(),
    PersistentDirectory.ice_staticId());
```

Next, we set all the `DirectoryI` and `FileI` static members.

```
Ice.ObjectAdapter adapter =
    communicator().createObjectAdapterWithEndpoints(
        "FreezeFilesystem", "default -p 10000");
DirectoryI._adapter = adapter;
FileI._adapter = adapter;

String dbName = "mapfs";
DirectoryI._communicator = communicator();
DirectoryI._envName = _envName;
DirectoryI._dbName = dbName;
FileI._communicator = communicator();
FileI._envName = _envName;
FileI._dbName = dbName;
```

Then we create a Freeze connection and a Freeze map. When the last connection to a Berkeley DB environment is closed, Freeze automatically closes this environment, so keeping a connection in the `main` function ensures the underlying Berkeley DB environment remains open. Likewise, we keep a map in the `main` function to keep the underlying Berkeley DB database open.

```
Freeze.Connection connection =
    Freeze.Util.createConnection(
        communicator(), _envName);
IdentityNodeMap persistentMap =
    new IdentityNodeMap(connection, dbName, true);
```

Now we need to initialize the root directory node. We first query the map for the identity of the root directory node; if no match is found, we create a new `PersistentDirectory` instance and insert it into the map.

```
Ice.Identity rootId =
    Ice.Util.stringToIdentity("RootDir");
PersistentDirectory pRoot =
    (PersistentDirectory)persistentMap.get(rootId);
if(pRoot == null)
{
    pRoot = new PersistentDirectory();
    pRoot.name = "/";
    pRoot.nodes = new java.util.HashMap();
    persistentMap.put(rootId, pRoot);
}
```

Finally, the `main` function instantiates the `Server` class, passing `db` as the name of the database environment.

```
public static void
main(String[] args)
{
    Server app = new Server("db");
    app.main("Server", args);
    System.exit(0);
}
```

The Servant Class Definitions

We also must change the servant classes to incorporate the Freeze map. We are maintaining the design from XREF, but we have added some methods and changed the constructor arguments and state members.

The `FileI` class has three new static members, `_communicator`, `_envName` and `_dbName`, that are used to instantiate the new `_connection` (a Freeze connection) and `_map` (an `IdentityNodeMap`) members in each `FileI` object. We keep the connection so that we can close it explicitly when we no longer need it.

The class also has a new instance member of type `PersistentFile`, representing the persistent state of this file, and a boolean member to indicate whether the node has been destroyed. Finally, we have changed its constructor to accept `Ice.Identity` and `PersistentFile`.

```
package Filesystem;

public class FileI extends _FileDisp
{
    public
    FileI(Ice.Identity id, PersistentFile file, DirectoryI parent)
    {
        // ...
    }

    // ... Ice operations ...

    public static Ice.ObjectAdapter _adapter;
    public static Ice.Communicator _communicator;
    public static String _envName;
    public static String _dbName;
    public Ice.Identity _ID;
    private Freeze.Connection _connection;
    private IdentityNodeMap _map;
```

```
    private PersistentFile _file;
    private DirectoryI _parent;
    private boolean _destroyed;
}
```

The `DirectoryI` class has undergone a similar transformation.

```
package Filesystem;

public final class DirectoryI extends _DirectoryDisp
{
    public
    DirectoryI(Ice.Identity id, PersistentDirectory dir,
               DirectoryI parent)
    {
        // ...
    }

    // ... Ice operations ...

    public static Ice.ObjectAdapter _adapter;
    public static Ice.Communicator _communicator;
    public static String _envName;
    public static String _dbName;
    public Ice.Identity _ID;
    private Freeze.Connection _connection;
    private IdentityNodeMap _map;
    private PersistentDirectory _dir;
    private DirectoryI _parent;
    private boolean _destroyed;
}
```

Implementing `FileI`

Let us examine how the implementations have changed. The `FileI` methods are still fairly trivial, but there are a few aspects that need discussion.

First, each operation now checks the `_destroyed` member and raises `Ice::ObjectNotExistException` if the member is `true`. This is necessary in order to ensure that the Freeze map is kept in a consistent state. For example, if we allowed the `write` operation to proceed after the file node had been destroyed, then we would have mistakenly added an entry back into the Freeze map for that file. Previous file system implementations ignored this issue because it was relatively harmless, but that is no longer true in this version.

Next, notice that the `write` operation calls `put` on the map after changing the `text` member of its `PersistentFile` object. Although this object is a value

in the Freeze map, changing the `text` member has no effect on the persistent state of the map until the value is reinserted into the map, thus overwriting the previous value.

Finally, the constructor now accepts an `Ice::Identity`. This differs from previous implementations in that the identity used to be computed by the constructor. In order to make our servants truly persistent, the identity for a node is computed once when that node is created, and must remain the same for the lifetime of the node. The `FileI` constructor therefore must not compute a new identity, but rather simply remember the identity that is given to it.

Similarly, the constructor is not responsible for creating a `PersistentFile` object, but rather is given one. This accommodates our two use cases: creating a new file, and restoring an existing file from the map.

```
public
FileI(Ice.Identity id, PersistentFile file,
       DirectoryI parent)
{
    _connection =
        Freeze.Util.createConnection(_communicator, _envName);
    _map =
        new IdentityNodeMap(_connection, _dbName, false);
    _ID = id;
    _file = file;
    _parent = parent;
    _destroyed = false;

    assert(_parent != null);

    // Add the identity of self to the object adapter
    //
    _adapter.add(this, _ID);
}

public synchronized String
name(Ice.Current current)
{
    if (_destroyed)
        throw new Ice.ObjectNotExistException();

    return _file.name;
}

public synchronized void
destroy(Ice.Current current)
```

```
        throws PermissionDenied
    {
        if (_destroyed)
            throw new Ice.ObjectNotExistException();

        _parent.removeChild(_file.name);
        _map.remove(current.id);
        current.adapter.remove(current.id);
        _map.close();
        _connection.close();
        _destroyed = true;
    }

    public synchronized String[]
read(Ice.Current current)
{
    if (_destroyed)
        throw new Ice.ObjectNotExistException();

    return _file.text;
}

public synchronized void
write(String[] text, Ice.Current current)
    throws GenericError
{
    if (_destroyed)
        throw new Ice.ObjectNotExistException();

    _file.text = text;
    _map.put(_ID, _file);
}
```

Implementing DirectoryI

The `DirectoryI` implementation requires more substantial changes. We begin our discussion with the `createDirectory` operation.

```
public synchronized DirectoryPrx
createDirectory(String name, Ice.Current current)
    throws NameInUse, IllegalName
{
    if (_destroyed)
        throw new Ice.ObjectNotExistException();

    checkName(name);
```

```

PersistentDirectory persistentDir
    = new PersistentDirectory();
persistentDir.name = name;
persistentDir.nodes = new java.util.HashMap();
DirectoryI dir = new DirectoryI(
    Ice.Util.stringToIdentity(Ice.Util.generateUUID()),
    persistentDir, this);
assert(_map.get(dir._ID) == null);
_map.put(dir._ID, persistentDir);

DirectoryPrx proxy = DirectoryPrxHelper.uncheckedCast(
    current.adapter.createProxy(dir._ID));

NodeDesc nd = new NodeDesc();
nd.name = name;
nd.type = NodeType.DirType;
nd.proxy = proxy;
_dir.nodes.put(name, nd);
_map.put(_ID, _dir);

return proxy;
}
}

```

After validating the node name, the operation creates a `PersistentDirectory`³ object for the child directory, which is passed to the `DirectoryI` constructor along with a unique identity. Next, we store the child's `PersistentDirectory` object in the Freeze map. Finally, we initialize a new `NodeDesc` value and insert it into the parent's node table and then reinsert the parent's `PersistentDirectory` object into the Freeze map.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```

public synchronized FilePrx
createFile(String name, Ice.Current current)
    throws NameInUse, IllegalName
{
    if (_destroyed)
        throw new Ice.ObjectNotExistException();
}

```

3. Since this Slice class has no operations, the compiler generates a concrete class that an application can instantiate.

```
    checkName(name);

    PersistentFile persistentFile = new PersistentFile();
    persistentFile.name = name;
    FileI file = new FileI(Ice.Util.stringToIdentity(
        Ice.Util.generateUUID()), persistentFile, this);
    assert(_map.get(file._ID) == null);
    _map.put(file._ID, persistentFile);

    FilePrx proxy = FilePrxHelper.uncheckedCast(
        current.adapter.createProxy(file._ID));

    NodeDesc nd = new NodeDesc();
    nd.name = name;
    nd.type = NodeType.FileType;
    nd.proxy = proxy;
    _dir.nodes.put(name, nd);
    _map.put(_ID, _dir);

    return proxy;
}
```

The next significant change is in the `DirectoryI` constructor. The body of the constructor now instantiates all of its immediate children, which effectively causes all nodes to be instantiated recursively.

For each entry in the directory's node table, the constructor locates the matching entry in the Freeze map. The key type of the Freeze map is `Ice::Identity`, so the constructor obtains the key by invoking `ice_getIdentity` on the child's proxy.

```
public
DirectoryI(Ice.Identity id, PersistentDirectory dir,
            DirectoryI parent)
{
    _connection =
        Freeze.Util.createConnection(_communicator, _envName);
    _map =
        new IdentityNodeMap(_connection, _dbName, false);
    _ID = id;
    _dir = dir;
    _parent = parent;
    _destroyed = false;

    // Add the identity of self to the object adapter
    //
```

```
_adapter.add(this, _ID);

// Instantiate the child nodes
//
java.util.Iterator p = dir.nodes.values().iterator();
while (p.hasNext()) {
    NodeDesc desc = (NodeDesc)p.next();
    Ice.Identity ident = desc.proxy.ice_getIdentity();
    PersistentNode node = (PersistentNode)_map.get(ident);
    assert(node != null);
    if (desc.type == NodeType.DirType) {
        PersistentDirectory pDir
            = (PersistentDirectory)node;
        DirectoryI d = new DirectoryI(ident, pDir, this);
    } else {
        PersistentFile pFile = (PersistentFile)node;
        FileI f = new FileI(ident, pFile, this);
    }
}
}
```

If it seems inefficient for the `DirectoryI` constructor to immediately instantiate all of its children, you are right. This clearly will not scale well to large node trees, so why are we doing it?

Previous implementations of the file system service returned transient proxies from `createDirectory` and `createFile`. In other words, if the server was stopped and restarted, any existing child proxies returned by the old instance of the server would no longer work. However, now that we have a persistent store, we should endeavor to ensure that proxies will remain valid across server restarts. There are a couple of implementation techniques that satisfy this requirement:

1. Instantiate all of the servants in advance, as shown in the `DirectoryI` constructor.
2. Use a servant locator.

We chose not to include a servant locator in this example because it complicates the implementation and, as we will see in Section 22.5, a Freeze evictor is ideally suited for this application and a better choice than writing a servant locator.

The last `DirectoryI` method we discuss is `removeChild`, which removes the entry from the node table, and then reinserts the `PersistentDirectory` object into the map to make the change persistent.

```
synchronized void  
removeChild(String name)  
{  
    _dir.nodes.remove(name);  
    _map.put(_ID, _dir);  
}
```

22.5 The Freeze Evictor

The Freeze evictor combines persistence and scalability features into a single facility that is easily incorporated into Ice applications.

As an implementation of the `ServantLocator` interface (see Section 16.6), the Freeze evictor takes advantage of the fundamental separation between Ice object and servant to activate servants on-demand from persistent storage, and to deactivate them again using customized eviction constraints. Although an application may have thousands of Ice objects in its database, it is not practical to have servants for all of those Ice objects resident in memory simultaneously. The application can conserve resources and gain greater scalability by setting an upper limit on the number of active servants, and letting the Freeze evictor handle the details of servant activation, persistence, and deactivation.

The Freeze evictor maintains a queue of active servants, ordered using a “least recently used” eviction algorithm: if the queue is full, the least recently used servant is evicted to make room for a new servant.

Here is the sequence of events for activating a servant as shown in Figure 22.3. Let us assume that we have configured the evictor with a size of five, that the queue is full, and that a request has arrived for a servant that is not currently active.

1. A client invokes an operation.
2. The object adapter invokes on the evictor to locate the servant.
3. The evictor first checks its active servant queue and fails to find the servant, so it instantiates the servant and restores its persistent state from the database.
4. The evictor adds an item for the servant (servant 1) at the head of the queue.
5. The queue’s length now exceeds the configured maximum, so the evictor removes servant 6 from the queue as soon as it is eligible for eviction. This occurs when there are no outstanding requests pending on servant 6, and its state has been safely stored in the database.

6. The object adapter dispatches the request to the new servant.

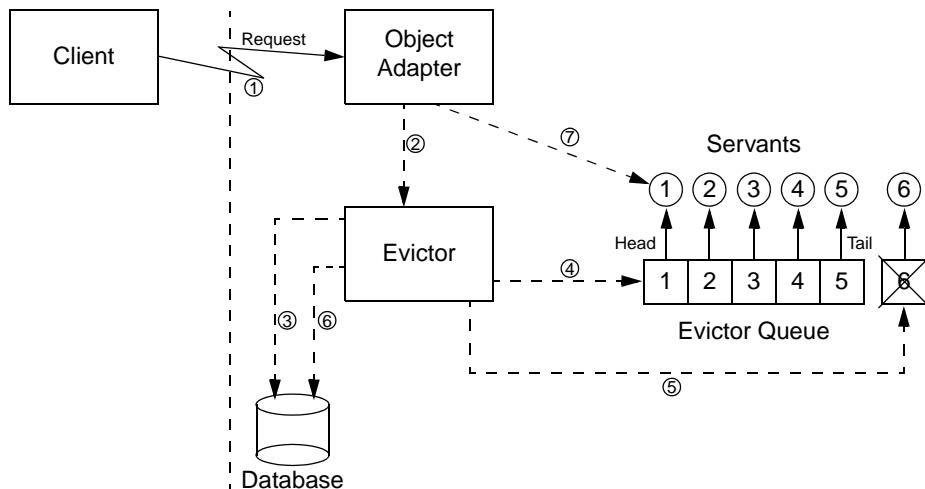


Figure 22.3. An evictor queue after restoring servant 1 and evicting servant 6.

22.5.1 Object Factories

The Freeze evictor is a generic facility in that it manages instances of `Object` subclasses. Applications are therefore free to use as many object types as necessary, with one requirement: an Ice object factory must be registered for each type.

22.5.2 Servant Association

With the Freeze evictor, each (object identity, facet) pair is associated with its own dedicated persistent object (servant). Such a persistent object cannot serve several identities or facets. Each servant is loaded and saved independently of other servants; in particular, there is no special grouping for the servants that serve the facets of a given Ice object.

Like the `ObjectAdapter`, the Freeze evictor provides operations named `add`, `addFacet`, `remove` and `removeFacet`. They have the same signature and semantics, except that with the Freeze evictor the mapping and the state of the mapped servants stored on in a database.

22.5.3 Detecting Updates

The Freeze Evictor considers that a servant has been modified when a mutating operation on this servant completes. (See Section 4.8 for more information on mutating operations.) Updates made internally, through calls that are not dispatched through the Freeze Evictor, are not detected and can be lost.

22.5.4 Saving Thread

All persistence activity of a Freeze evictor is handled in a background thread created by the evictor. This thread wakes up periodically and saves the state of all newly-registered, modified, and destroyed servant in the evictor's queue.

For applications that experience bursts of activity, resulting in a large number of modified servants in a short period of time, the evictor's thread can also be configured to begin saving as soon as the number of modified servants reaches a certain threshold.

22.5.5 Synchronization

When the saving thread takes a snapshot of a servant it is about to save, it is necessary to prevent the application from modifying the servant's persistent data members at the same time.

The Freeze evictor and the application need to use a common synchronization to ensure correct behavior. In Java, this common synchronization is the servant itself: the Freeze evictor synchronizes the servant (a Java object) while taking the snapshot. In C++, the servant is required to inherit from the class

`IceUtil::AbstractMutex`: the Freeze evictor locks the servant through this interface while taking a snapshot. On the application side, the servant's implementation is required to synchronize all operations that access the servant's data members defined in Slice using the same mechanism.

22.5.6 Index

The Freeze evictor supports the use of indices to quickly find persistent objects using the value of a data member as the search criteria. The types allowed for these indices are the same as the types allowed for Slice dictionary keys (see Section 4.7.4).

The `slice2freeze` and `slicefreezej` tools can generate an `Index` class when passed the `--index` option:

- **--index CLASS,TYPE, MEMBER[,case-sensitive|case-insensitive]**

CLASS is the name of the class to be generated. **TYPE** denotes the type of class to be indexed (objects of different classes are not included in this index). **MEMBER** is the name of the data member in **TYPE** to index. When **MEMBER** has type **string**, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

The generated `Index` class supplies three methods:

- `sequence<Ice::Identity> findFirst(member-type index, int firstN)`
Returns up to `firstN` objects of **TYPE** whose **MEMBER** is equal to `index`. This is useful to avoid running out of memory if the number of objects matching the criteria can be very large.
- `sequence<Ice::Identity> find(member-type index)`
Returns all the objects of **TYPE** whose **MEMBER** is equal to `index`.
- `int count(<type> index)`
Returns the number of objects of **TYPE** having **MEMBER** equal to `index`.

Indices are associated with a Freeze evictor during evictor creation. See the definition of the `createEvictor` methods for details.

Indexed searches are easy to use and very efficient. However, be aware that an index adds significant write overhead: with Berkeley DB, every single update triggers a read from the database to get the old index entry and, if necessary, replace it.

22.5.7 Using a Servant Initializer

In some applications it may be necessary to initialize a servant after it is instantiated by the evictor but before an operation is dispatched to it. The Freeze evictor allows an application to specify a servant initializer for this purpose.

To clarify the sequence of events, let us assume that a request has arrived for an Ice facet that is not currently active:

1. The evictor restores a servant for the Ice facet from the database. This involves two steps:
 1. The Ice run time locates and invokes the factory for the Ice facet's type, thereby obtaining a new instance with uninitialized data members.
 2. The data members are populated from the persistent state.

2. The evictor invokes the application's servant initializer (if any) for the servant.
3. The evictor adds the servant to its cache.
4. The evictor dispatches the operation.

The servant initializer is called before the object is inserted into the Freeze evictor internal cache, and *without* holding any internal lock, but in such a way that when the servant initializer is called, the servant is guaranteed to be inserted in the Freeze evictor cache.

There is only one restriction on what a servant initializer can do: it must not make a remote invocation on the facet being initialized. Failing to follow this rule will result in deadlocks.

The file system implementation presented in Section 22.6 on page 636 demonstrates the use of a servant initializer.

22.5.8 Application Design Considerations

The Freeze evictor creates a snapshot of an Ice facet's state for persistent storage by marshaling the facet, just as if the facet were being sent "over the wire" as a parameter to a remote invocation. Therefore, the Slice definitions for an object type must include the data members comprising the object's persistent state.

For example, we could define a Slice class as follows:

```
class Stateless {  
    void calc();  
};
```

However, without data members, there will not be any persistent state in the database for objects of this type, and hence there is little value in using the Freeze evictor for this type.

Obviously, Slice object types need to define data members, but there are other design considerations as well. For example, suppose we define a simple application as follows:

```
class Account {  
    void withdraw(int amount);  
    void deposit(int amount);  
  
    int balance;  
};
```

```
interface Bank {  
    Account* createAccount();  
};
```

In this application, we would use the Freeze evictor to manage `Account` objects that have a data member `balance` representing the persistent state of an account.

From an object-oriented design perspective, there is a glaring problem with these Slice definitions: implementation details (the persistent state) are exposed in the client-server contract. The client cannot directly manipulate the `balance` member because the `Bank` interface returns `Account` proxies, not `Account` instances. However, the presence of the data member may cause unnecessary confusion for client developers.

A better alternative is to clearly separate the persistent state as shown below:

```
interface Account {  
    void withdraw(int amount);  
    void deposit(int amount);  
};  
  
interface Bank {  
    Account* createAccount();  
};  
  
class PersistentAccount implements Account {  
    int balance;  
};
```

Now the Freeze evictor can manage `PersistentAccount` objects, while clients interact with `Account` proxies. (Ideally, `PersistentAccount` would be defined in a different source file and inside a separate module.)

22.6 Using the Freeze Evictor in a File System Server

In this section, we present file system implementations that utilize a Freeze evictor. The implementations are based on the ones discussed in XREF, and in this section we only discuss code that illustrates use of the Freeze evictor.

In general, incorporating a Freeze evictor into your application requires the following steps:

1. Evaluate your existing Slice definitions for a suitable persistent object type.

2. If no suitable type is found, you typically define a new derived class which captures your persistent state requirements. Consider placing these definitions in a separate file: they are only used by the server for persistence, and therefore do not need to appear in the “public” definitions required by clients. Also consider placing your persistent types in a separate module to avoid name clashes.
3. Generate code (using **slice2freeze** or **slice2freezej**) for your new definitions.
4. Create an evictor and register it as a servant locator with an object adapter.
5. Create instances of your persistent type and register them with the evictor.

22.6.1 Slice Definitions

Fortunately, it is unnecessary for us to change any of the existing file system Slice definitions to incorporate the Freeze evictor. However, we do need to add some definitions to express our persistent state requirements:

```
module Filesystem {  
    class PersistentDirectory;  
  
    class PersistentNode implements Node {  
        string nodeName;  
        PersistentDirectory* parent;  
    };  
  
    class PersistentFile extends PersistentNode implements File {  
        Lines text;  
    };  
  
    class PersistentDirectory extends PersistentNode  
        implements Directory {  
        void removeNode(string name);  
  
        NodeDict nodes;  
    };  
};
```

As you can see, we have subclassed all of the node interfaces. Let us examine each one in turn.

The `PersistentNode` class adds two data members: `nodeName`⁴ and `parent`. The file system implementation requires that a child node know its parent node in order to properly implement the `destroy` operation. Previous implementations

had a state member of type `DirectoryI`, but that is not workable here. It is no longer possible to pass the parent node to the child node's constructor because the evictor may be instantiating the child node (via a factory), and the parent node will not be known. Even if it were known, another factor to consider is that there is no guarantee that the parent node will be active when the child invokes on it, because the evictor may have evicted it. We solve these issues by storing a proxy to the parent node. If the child node invokes on the parent node via the proxy, the evictor automatically activates the parent node if necessary.

The `PersistentFile` class is very straightforward, simply adding a `text` member representing the contents of the file. Notice that the class extends `PersistentNode`, and therefore inherits the state members declared by the base class.

Finally, the `PersistentDirectory` class defines the `removeNode` operation, and adds the `nodes` state member representing the immediate children of the directory node. Since a child node contains only a proxy for its `PersistentDirectory` parent, and not a reference to an implementation class, there must be a Slice-defined operation that can be invoked when the child is destroyed.

If we had followed the advice at the beginning of Section 22.5, we would have defined `Node`, `File`, and `Directory` classes in a separate `PersistentFilesystem` module, but in this example we use the existing `Filesystem` module for the sake of simplicity.

22.6.2 Implementing the File System Server in C++

The Server `main` Program

The server's `main` program is responsible for creating the evictor and initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice::Application` as described in Section 10.3.1. Our server main program has now become the following:

4. We used `nodeName` instead of `name` because `name` is already used as an operation in the `Node` interface.

```
#include <FilesystemI.h>
#include <Freeze/Freeze.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : virtual public Ice::Application {
public:
    FilesystemApp(const string & envName) :
        _envName(envName) { }

    virtual int run(int, char * []) {
        // Install object factories
        //
        Ice::ObjectFactoryPtr factory = new NodeFactory;
        communicator()->addObjectFactory(
            factory,
            PersistentFile::ice_staticId());
        communicator()->addObjectFactory(
            factory,
            PersistentDirectory::ice_staticId());

        // Create an object adapter (stored in the NodeI::_adapter
        // static member)
        //
        NodeI::_adapter =
            communicator()->createObjectAdapterWithEndpoints(
                "FreezeFilesystem", "default -p 10000");

        // Create the Freeze evictor (stored in the
        // NodeI::_evictor static member)
        //
        Freeze::ServantInitializerPtr init = new NodeInitializer;
        NodeI::_evictor =
            Freeze::createEvictor(NodeI::_adapter, _envName,
                "evictorfs", init);

        NodeI::_adapter->addServantLocator(NodeI::_evictor, "");

        // Create the root node if it doesn't exist
        //
        Ice::Identity rootId = Ice::stringToIdentity("RootDir");
        if (!NodeI::_evictor->hasObject(rootId)) {
            PersistentDirectoryPtr root = new DirectoryI(rootId);
            root->nodeName = "/";
            NodeI::_evictor->add(root, rootId);
        }
    }
}
```

```
    }

    // Ready to accept requests now
    //
    NodeI::_adapter->activate();

    // Wait until we are done
    //
    communicator()->waitForShutdown();
    if (interrupted()) {
        cerr << appName()
            << ": received signal, shutting down" << endl;
    }

    return 0;
}

private:
    string _envName;
};

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}
```

Let us examine the changes in detail. First, we are now including `Freeze/Freeze.h` instead of `Ice/Ice.h`. This `Freeze` header file includes all of the other `Freeze` (and `Ice`) header files this source file requires.

Next, we define the class `FilesystemApp` as a subclass of `Ice::Application`, and provide a constructor taking a string argument:

```
FilesystemApp(const string & envName) :
    _envName(envName) { }
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

One of the first tasks `run` performs is installing the `Ice` object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. A single instance of `NodeFactory` is installed for both types.

```
Ice::ObjectFactoryPtr factory = new NodeFactory;
communicator()-> addObjectFactory(
    factory,
    PersistentFile::ice_staticId());
communicator()-> addObjectFactory(
    factory,
    PersistentDirectory::ice_staticId());
```

After creating the object adapter, the program initializes a Freeze evictor by invoking `createEvictor`. The third argument to `createEvictor` is the name of the database file, which by default is created if it does not exist. The new evictor is then added to the object adapter as a servant locator for the default category.

```
Freeze::ServantInitializerPtr init = new NodeInitializer;
NodeI::_evictor =
    Freeze::createEvictor(NodeI::_adapter, _envName,
                         "evictorfs", init);
NodeI::_adapter->addServantLocator(NodeI::_evictor, "");
```

Next, the program creates the root directory node if it is not already being managed by the evictor.

```
Ice::Identity rootId = Ice::stringToIdentity("RootDir");
if (!NodeI::_evictor->hasObject(rootId)) {
    PersistentDirectoryPtr root = new DirectoryI(rootId);
    root->nodeName = "/";
    NodeI::_evictor->add(root, rootId);
}
```

Finally, the `main` function instantiates the `FilesystemApp`, passing `db` as the name of the database environment.

```
int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}
```

The Servant Class Definitions

The servant classes must also be changed to incorporate the Freeze evictor. We are maintaining the multiple-inheritance design from XREF, but we have changed the constructors and state members. In particular, each node implementation class has two constructors, one taking no parameters and one taking an `Ice::Identity`.

The former is needed by the factory, and the latter is used when the node is first created. To comply with the evictor requirements, `NodeI` implements `IceUtil::AbstractMutex` by deriving from the template class `IceUtil::AbstractMutexI`. The only other change of interest is a new state member in `NodeI` named `_evictor`.

```
        DirectoryI();
        DirectoryI(const Ice::Identity &);

protected:
    void listRecursive(const std::string &,
                      const DirectoryPrx &,
                      NodeDict &) const;

private:
    void checkName(const std::string &) const;
};

}
```

In addition to the node implementation classes, we have also declared implementations of an object factory and a servant initializer:

```
namespace Filesystem {
    class NodeFactory : virtual public Ice::ObjectFactory {
public:
    virtual Ice::ObjectPtr create(const std::string &);
    virtual void destroy();
};

    class NodeInitializer :
        virtual public Freeze::ServantInitializer {
public:
    virtual void initialize(const Ice::ObjectAdapterPtr &,
                           const Ice::Identity &,
                           const std::string &,
                           const Ice::ObjectPtr &);
};

}
```

Implementing NodeI

Notice that the `NodeI` constructor no longer computes a value for the identity. This is necessary in order to make our servants truly persistent. Specifically, the identity for a node is computed once when that node is created, and must remain the same for the lifetime of the node. The `NodeI` constructor therefore must not compute a new identity, but rather remember the identity that is given to it.

```
string
Filesystem::NodeI::name(const Ice::Current &) const
{
    return nodeName;
}

void
```

```
Filesystem::NodeI::destroy(const Ice::Current & current)
{
    if (!parent) {
        Filesystem::PermissionDenied e;
        e.reason = "cannot remove root directory";
        throw e;
    }

    parent->removeNode(nodeName);
    _evictor->remove(_ID);
}

Filesystem::NodeI::NodeI()
{
}

Filesystem::NodeI::NodeI(const Ice::Identity & id)
    : _ID(id)
{
}
```

Implementing FileI

The FileI methods are still fairly trivial, because the Freeze evictor is handling persistence for us.

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    IceUtil::Mutex::Lock lock(*this);

    return text;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                       const Ice::Current &)
{
    IceUtil::Mutex::Lock lock(*this);

    this->text = text;
}

Filesystem::FileI::FileI()
{
}
```

```
Filesystem::FileI::FileI(const Ice::Identity & id)
    : NodeI(id)
{
}
```

Implementing DirectoryI

The `DirectoryI` implementation requires more substantial changes. We begin our discussion with the `createDirectory` operation.

```
Filesystem::DirectoryPrx
Filesystem::DirectoryI::createDirectory(
    const string & name,
    const Ice::Current & current)
{
    IceUtil::Mutex::Lock lock(*this);

    checkName(name);

    Ice::Identity id =
        Ice::stringToIdentity(IceUtil::generateUUID());
    PersistentDirectoryPtr dir = new DirectoryI(id);
    dir->nodeName = name;
    dir->parent = PersistentDirectoryPrx::uncheckedCast(
        current.adapter->createProxy(current.id));
    DirectoryPrx proxy = _evictor->add(dir, id);

    NodeDesc nd;
    nd.name = name;
    nd.type = DirType;
    nd.proxy = proxy;
    nodes[name] = nd;

    return proxy;
}
```

After validating the node name, the operation obtains a unique identity for the child directory, instantiates the servant, and registers it with the Freeze evictor. Finally, the operation creates a proxy for the child and adds the child to its node table.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```
Filesystem::FilePrx
Filesystem::DirectoryI::createFile(
    const string & name,
    const Ice::Current & current)
{
    IceUtil::Mutex::Lock lock(*this);

    checkName(name);

    Ice::Identity id =
        Ice::stringToIdentity(IceUtil::generateUUID());
    PersistentFilePtr file = new FileI(id);
    file->nodeName = name;
    file->parent = PersistentDirectoryPrx::uncheckedCast(
        current.adapter->createProxy(current.id));
    FilePrx proxy = _evictor->add(file, id);

    NodeDesc nd;
    nd.name = name;
    nd.type = FileType;
    nd.proxy = proxy;

    return proxy;
}
```

Implementing NodeFactory

We use a single factory implementation for creating two types of Ice objects: `PersistentFile` and `PersistentDirectory`. These are the only two types that the Freeze evictor will be restoring from its database.

```
Ice::ObjectPtr
Filesystem::NodeFactory::create(const string & type)
{
    if (type == "::Filesystem::PersistentFile")
        return new FileI;
    else if (type == "::Filesystem::PersistentDirectory")
        return new DirectoryI;
    else {
        assert(false);
        return 0;
    }
}
```

```
void
Filesystem::NodeFactory::destroy()
{
}
```

Implementing NodeInitializer

`NodeInitializer` is a trivial implementation of the `Freeze::ServantInitializer` interface whose only responsibility is setting the `_ID` member of the node implementation. The evictor invokes the `initialize` operation after the evictor has created a servant and restored its persistent state from the database, but before any operations are dispatched to it.

```
void
Filesystem::NodeInitializer::initialize(
    const Ice::ObjectAdapterPtr &,
    const Ice::Identity & id,
    const Ice::ObjectPtr & obj)
{
    NodeIPtr node = NodeIPtr::dynamicCast(obj);
    assert(node);
    const_cast<Ice::Identity&>(node->_ID) = id;
}
```

22.6.3 Implementing the File System Server in Java

The Server main Program

The server's main program is responsible for creating the evictor and initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice.Application` as described in Section 12.3.1. Our server main program has now become the following:

```
import Filesystem.*;

public class Server extends Ice.Application {
    public
    Server(String envName)
    {
        _envName = envName;
    }

    public int
    run(String[] args)
```

```
{  
    // Install object factories  
    //  
    Ice.ObjectFactory factory = new NodeFactory();  
    communicator().addObjectFactory(  
        factory,  
        PersistentFile.ice_staticId());  
    communicator().addObjectFactory(  
        factory,  
        PersistentDirectory.ice_staticId());  
  
    // Create an object adapter (stored in the _adapter  
    // static member)  
    //  
    Ice.ObjectAdapter adapter =  
        communicator().createObjectAdapterWithEndpoints(  
            "FreezeFilesystem", "default -p 10000");  
    DirectoryI._adapter = adapter;  
    FileI._adapter = adapter;  
  
    // Create the Freeze evictor (stored in the _evictor  
    // static member)  
    //  
    Freeze.ServantInitializer init = new NodeInitializer();  
    Freeze.Evictor evictor =  
        Freeze.Util.createEvictor(adapter, _envName,  
            "evictorfs", init, null,  
            true);  
    DirectoryI._evictor = evictor;  
    FileI._evictor = evictor;  
  
    adapter.addServantLocator(evictor, "");  
  
    // Create the root node if it doesn't exist  
    //  
    Ice.Identity rootId =  
        Ice.Util.stringToIdentity("RootDir");  
    if(!evictor.hasObject(rootId))  
    {  
        PersistentDirectory root = new DirectoryI(rootId);  
        root.nodeName = "/";  
        root.nodes = new java.util.HashMap();  
        evictor.add(root, rootId);  
    }  
  
    // Ready to accept requests now
```

```
//  
adapter.activate();  
  
// Wait until we are done  
//  
communicator().waitForShutdown();  
  
    return 0;  
}  
  
public static void  
main(String[] args)  
{  
    Server app = new Server("db");  
    app.main("Server", args);  
    System.exit(0);  
}  
  
private String _envName;  
}
```

Let us examine the changes in detail. First, we define the class `Server` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

```
public  
Server(String envName)  
{  
    _envName = envName;  
}
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

One of the first tasks `run` performs is installing the Ice object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. A single instance of `NodeFactory` is installed for both types.

```
Ice.ObjectFactory factory = new NodeFactory();  
communicator().addObjectFactory(  
    factory,  
    PersistentFile.ice_staticId());  
communicator().addObjectFactory(  
    factory,  
    PersistentDirectory.ice_staticId());
```

After creating the object adapter, the program initializes a Freeze evictor by invoking `createEvictor`. The third argument to `createEvictor` is the name of the database, the fourth is the servant initializer, then the `null` argument indicates no indices are in use, and the `true` argument requests that the database be created if it does not exist. The evictor is then added to the object adapter as a servant locator for the default category.

```
Freeze.ServantInitializer init = new NodeInitializer();
Freeze.Evictor evictor =
    Freeze.Util.createEvictor(adapter, _envName,
                               "evictorfs", init, null,
                               true);
DirectoryI._evictor = evictor;
FileI._evictor = evictor;

adapter.addServantLocator(evictor, "");
```

Next, the program creates the root directory node if it is not already being managed by the evictor.

```
Ice.Identity rootId =
    Ice.Util.stringToIdentity("RootDir");
if(!evictor.hasObject(rootId))
{
    PersistentDirectory root = new DirectoryI(rootId);
    root.nodeName = "/";
    root.nodes = new java.util.HashMap();
    evictor.add(root, rootId);
}
```

Finally, the `main` function instantiates the `Server` class, passing `db` as the name of the database environment.

```
public static void
main(String[] args)
{
    Server app = new Server("db");
    app.main("Server", args);
    System.exit(0);
}
```

The Servant Class Definitions

The servant classes must also be changed to incorporate the Freeze evictor. We are maintaining the design from XREF, but we have changed the constructors and state members. In particular, each node implementation class has two construc-

tors, one taking no parameters and one taking an `Ice::Identity`. The former is needed by the factory, and the latter is used when the node is first created. The only other change of interest is the new state member `_evictor`.

The `FileI` class has a new static member of type `IdentityNodeMap`, which avoids the need to duplicate this reference in each node. The class has a new instance member of type `PersistentFile`, representing the persistent state of this file. Finally, we have changed the constructor to accept an `Ice.Identity` and a `PersistentFile`.

```
package Filesystem;

public class FileI extends PersistentFile
{
    public
    FileI()
    {
        // ...
    }

    public
    FileI(Ice.Identity id)
    {
        // ...
    }

    // ... Ice operations ...

    public static Ice.ObjectAdapter _adapter;
    public static Freeze.Evictor _evictor;
    public Ice.Identity _ID;
}
```

The `DirectoryI` class has undergone a similar transformation.

```
package Filesystem;

public final class DirectoryI extends PersistentDirectory
{
    public
    DirectoryI()
    {
        // ...
    }

    public
```

```
DirectoryI(Ice.Identity id)
{
    // ...
}

// ... Ice operations ...

public static Ice.ObjectAdapter _adapter;
public static Freeze.Evictor _evictor;
public Ice.Identity _ID;
}
```

Notice that the constructors no longer compute a value for the identity. This is necessary in order to make our servants truly persistent. Specifically, the identity for a node is computed once when that node is created, and must remain the same for the lifetime of the node. A constructor therefore must not compute a new identity, but rather remember the identity given to it.

Implementing FileI

The FileI methods are still fairly trivial, because the Freeze evictor is handling persistence for us.

```
public
FileI()
{
}

public
FileI(Ice.Identity id)
{
    _ID = id;
}

public String
name(Ice.Current current)
{
    return nodeName;
}

public void
destroy(Ice.Current current)
    throws PermissionDenied
{
    parent.removeNode(nodeName);
    _evictor.remove(current.id);
```

```
}

public synchronized String[]
read(Ice.Current current)
{
    return text;
}

public synchronized void
write(String[] text, Ice.Current current)
throws GenericError
{
    this.text = text;
}
```

Implementing DirectoryI

The `DirectoryI` implementation requires more substantial changes. We begin our discussion with the `createDirectory` operation.

```
public synchronized DirectoryPrx
createDirectory(String name, Ice.Current current)
throws NameInUse, IllegalName
{
    checkName(name);

    Ice.Identity id =
        Ice.Util.stringToIdentity(Ice.Util.generateUUID());
    PersistentDirectory dir = new DirectoryI(id);
    dir.nodeName = name;
    dir.parent = PersistentDirectoryPrxHelper.uncheckedCast(
        current.adapter.createProxy(current.id));
    dir.nodes = new java.util.HashMap();
    DirectoryPrx proxy = _evictor.add(dir, id);

    NodeDesc nd = new NodeDesc();
    nd.name = name;
    nd.type = NodeType.DirType;
    nd.proxy = proxy;
    nodes.put(name, nd);

    return proxy;
}
```

After validating the node name, the operation obtains a unique identity for the child directory, instantiates the servant, and registers it with the Freeze evictor.

Finally, the operation creates a proxy for the child and adds the child to its node table.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```
public synchronized FilePrx
createFile(String name, Ice.Current current)
    throws NameInUse, IllegalName
{
    checkName(name);

    Ice.Identity id =
        Ice.Util.stringToIdentity(Ice.Util.generateUUID());
    PersistentFile file = new FileI(id);
    file.nodeName = name;
    file.parent = PersistentDirectoryPrxHelper.uncheckedCast(
        current.adapter.createProxy(current.id));
    FilePrx proxy = _evictor.add(file, id);

    NodeDesc nd = new NodeDesc();
    nd.name = name;
    nd.type = NodeType.FileType;
    nd.proxy = proxy;
    nodes.put(name, nd);

    return proxy;
}
```

Implementing `NodeFactory`

We use a single factory implementation for creating two types of Ice objects: `PersistentFile` and `PersistentDirectory`. These are the only two types that the Freeze evictor will be restoring from its database.

```
package Filesystem;

public class NodeFactory extends Ice.LocalObjectImpl
    implements Ice.ObjectFactory
{
    public Ice.Object
    create(String type)
    {
        if (type.equals "::Filesystem::PersistentFile")
            return new FileI();
        else if (type.equals "::Filesystem::PersistentDirectory")
            return new DirectoryI();
```

```
        else {
            assert(false);
            return null;
        }
    }

    public void
    destroy()
    {
    }
}
```

Implementing NodeInitializer

`NodeInitializer` is a trivial implementation of the `Freeze::ServantInitializer` interface whose only responsibility is setting the `_ID` member of the node implementation. The evictor invokes the `initialize` operation after the evictor has created a servant and restored its persistent state from the database, but before any operations are dispatched to it.

```
package Filesystem;

public class NodeInitializer extends Ice.LocalObjectImpl
    implements Freeze.ServantInitializer {
    public void
    initialize(Ice.ObjectAdapter adapter, Ice.Identity id,
               String facet, Ice.Object obj)
    {
        if (obj instanceof FileI)
            ((FileI)obj).ID = id;
        else
            ((DirectoryI)obj).ID = id;
    }
}
```

22.7 Summary

Freeze is a collection of services that simplify the use of persistence in Ice applications. The Freeze map is an associative container mapping any Slice key and value types, providing a convenient and familiar interface to a persistent map. The Freeze evictor is an especially powerful facility for supporting persistent Ice objects in a highly-scalable implementation.

Chapter 23

FreezeScript

23.1 Chapter Overview

This chapter describes the FreezeScript tools for migrating and inspecting the databases created by Freeze maps and evictors. The discussion of database migration begins in Section 23.3 and continues through Section 23.5. Database inspection is presented in Section 23.6 and Section 23.7. Finally, Section 23.8 describes the expression language supported by the FreezeScript tools.

23.2 Introduction

As described in Chapter 22, Freeze supplies a valuable set of services for simplifying the use of persistence in Ice applications. However, while Freeze makes it easy for an application to manage its persistent state, there are additional administrative responsibilities that must also be addressed:

- Migration

As an application evolves, it is not unusual for the types describing its persistent state to evolve as well. When these changes occur, a great deal of time can be saved if existing databases can be migrated to the new format while preserving as much information as possible.

- Inspection

The ability to examine a database can be helpful during every stage of the application’s lifecycle, from development to deployment.

FreezeScript provides tools for performing both of these activities on Freeze map and evictor databases. These databases have a well-defined structure because the key and value of each record consist of the marshaled bytes of their respective Slice types. This design allows the FreezeScript tools to operate on any Freeze database using only the Slice definitions for the database types.

23.3 Database Migration

The FreezeScript tool **transformdb** migrates a database created by a Freeze map or evictor. It accomplishes this by comparing the “old” Slice definitions (i.e., the ones that describe the current contents of the database) with the “new” Slice definitions, and making whatever modifications are necessary to ensure that the transformed database is compatible with the new definitions.

This would be difficult to achieve by writing a custom transformation program because that program would require static knowledge of the old and new types, which frequently define many of the same symbols and would therefore prevent the program from being loaded. The **transformdb** tool avoids this issue using an interpretive approach: the Slice definitions are parsed and used to drive the extraction of the database records.

The tool supports two modes of operation:

1. automatic migration, in which the database is migrated in a single step using only the default set of transformations, and
2. custom migration, in which you supply a script to augment or override the default transformations.

23.3.1 Default Transformations

The default transformations performed by **transformdb** preserve as much information as possible. However, there are practical limits to the tool’s capabilities, since the only information it has is obtained by performing a comparison of the Slice definitions.

For example, suppose our old definition for a structure is the following:

```
struct AStruct {  
    int i;  
};
```

We want to migrate instances of this struct to the following revised definition:

```
struct AStruct {  
    int j;  
};
```

As the developers, we know that the `int` member has been renamed from `i` to `j`, but to `transformdb` it appears that member `i` was removed and member `j` was added. The default transformation results in exactly that behavior: the value of `i` is lost, and `j` is initialized to a default value. If we need to preserve the value of `i` and transfer it to `j`, then we need to use custom migration (see Section 23.3.5).

The changes that occur as a type system evolves can be grouped into three categories:

- Data members

The data members of class and structure types are added, removed, or renamed. As discussed above, the default transformations initialize new and renamed data members to default values (see Section 23.3.3).

- Type names

Types are added, removed, or renamed. New types do not pose a problem for database migration when used to define a new data member; the member is initialized with default values as usual. On the other hand, if the new type replaces the type of an existing data member, then type compatibility becomes a factor (see the following item).

Removed types generally do not cause problems either, because any uses of that type must have been removed from the new Slice definitions (e.g., by removing data members of that type). There is one case, however, where removed types become an issue, and that is for polymorphic classes (see Section 23.5.5).

Renamed types are a concern, just like renamed data members, because of the potential for losing information during migration. This is another situation for which custom migration is recommended.

- Type content

Examples of changes of type content include the key type of a dictionary, the element type of a sequence, or the type of a data member. If the old and new types are not compatible (as defined in Section 23.3.2), then the default trans-

formation emits a warning, discards the current value, and reinitializes the value as described in Section 23.3.3.

23.3.2 Type Compatibility

Changes in the type of a value are restricted to certain sets of compatible changes. This section describes the type changes supported by the default transformations. All incompatible type changes result in a warning indicating that the current value is being discarded and a default value for the new type assigned in its place. Additional flexibility is provided by custom migration, as described in Section 23.3.5.

Boolean

A value of type `bool` can be transformed to and from `string`. The legal string values for a `bool` value are "`true`" and "`false`".

Integer

The integer types `byte`, `short`, `int`, and `long` can be transformed into each other, but only if the current value is within range of the new type. These integer types can also be transformed into `string`.

Floating Point

The floating-point types `float` and `double` can be transformed into each other, as well as to `string`. No attempt is made to detect a loss of precision during transformation.

String

A `string` value can be transformed into any of the primitive types, as well as into enumeration and proxy types, but only if the value is a legal string representation of the new type. For example, the string value "`Pear`" can be transformed into the enumeration `Fruit`, but only if `Pear` is an enumerator of `Fruit`.

Enum

An enumeration can be transformed into an enumeration with the same type id, or into a string. Transformation between enumerations is performed symbolically. For example, consider our old type below:

```
enum Fruit { Apple, Orange, Pear };
```

Suppose the enumerator `Pear` is being transformed into the following new type:

```
enum Fruit { Apple, Pear };
```

The transformed value in the new enumeration is also `Pear`, despite the fact that `Pear` has changed positions in the new type. However, if the old value had been `Orange`, then the default transformation emits a warning because that enumerator no longer exists, and initializes the new value to `Apple` (the default value).

If an enumerator has been renamed, then custom migration is required to convert enumerators from the old name to the new one.

Sequence

A sequence can be transformed into another sequence type, even if the new sequence type does not have the same type id as the old type, but only if the element types are compatible. For example, `sequence<short>` can be transformed into `sequence<int>`, regardless of the names given to the sequence types.

Dictionary

A dictionary can be transformed into another dictionary type, even if the new dictionary type does not have the same type id as the old type, but only if the key and value types are compatible. For example, `dictionary<int, string>` can be transformed into `dictionary<long, string>`, regardless of the names given to the dictionary types.

Caution is required when changing the key type of a dictionary, because the default transformation of keys could result in duplication. For example, if the key type changes from `int` to `short`, any `int` value outside the range of `short` results in the key being initialized to a default value (namely zero). If zero is already used as a key in the dictionary, or another out-of-range key is encountered, then a duplication occurs. The transformation handles key duplication by removing the duplicate element from the transformed dictionary. (Custom migration can be useful in these situations if the default behavior is not acceptable.)

Structure

A `struct` type can only be transformed into another `struct` type with the same type id. Data members are transformed as appropriate for their types.

Proxy

A proxy value can be transformed into another proxy type, or into `string`. Transformation into another proxy type is done with the same semantics as in a

language mapping: if the new type does not match the old type, then the new type must be a base type of the old type (that is, the proxy is widened).

Class

A `class` type can only be transformed into another `class` type with the same type id. A data member of a `class` type is allowed to be widened to a base type. Data members are transformed as appropriate for their types. See Section 23.5.5 for more information on transforming classes.

23.3.3 Default Values

Data types are initialized with default values, as shown in Table 23.1.

Table 23.1. Default values for Slice types.

Type	Default Value
Boolean	<code>false</code>
Numeric	Zero (0)
String	Empty string
Enumeration	The first enumerator
Sequence	Empty sequence
Dictionary	Empty dictionary
Struct	Data members are initialized recursively
Proxy	Nil
Class	Nil

23.3.4 Running an Automatic Transformation

In order to use automatic transformation, we need to supply the following information to `transformdb`:

- The old and new Slice definitions
- The old and new types for the database key and value
- The database environment directory, the database filename, and the name of a new database environment directory to hold the transformed database

Here is an example of a `transformdb` command:

```
$ transformdb --old old/MyApp.ice --new new/MyApp.ice \
  --key int,string --value ::Employee db emp.db newdb
```

Briefly, the **--old** and **--new** options specify the old and new Slice definitions, respectively. These options can be specified as many times as necessary in order to load all of the relevant definitions. The **--key** option indicates that the database key is evolving from `int` to `string`. The **--value** option specifies that `::Employee` is used as the database value type in both old and new type definitions, and therefore only needs to be specified once. Finally, we provide the pathname of the database environment directory (`db`), the filename of the database (`emp.db`), and the pathname of the database environment directory for the transformed database (`newdb`).

See Section 23.5 for more information on using `transformdb`.

23.3.5 Custom Migration

Custom migration is useful when your types have changed in ways that make automatic migration difficult or impossible. It is also convenient to use custom migration when you have complex initialization requirements for new types or new data members, because custom migration enables you to perform many of the same tasks that would otherwise require you to write a throwaway program.

Custom migration operates in conjunction with automatic migration, allowing you to inject your own transformation rules at well-defined intercept points in the automatic migration process. These rules are called *transformation descriptors*, and are written in XML.

A Simple Example

We can use a simple example to demonstrate the utility of custom migration. Suppose our application uses a Freeze map whose type is `string` and whose value is an enumeration, defined as follows:

```
enum BigThree { Ford, Chrysler, GeneralMotors };
```

We now wish to rename the enumerator `Chrysler`, as shown in our new definition:

```
enum BigThree { Ford, DaimlerChrysler, GeneralMotors };
```

As explained in Section 23.3.2, the default transformation results in all occurrences of the `Chrysler` enumerator being transformed into `Ford`, because

`Chrysler` no longer exists in the new definition and therefore the default value `Ford` is used instead.

To remedy this situation, we use the following transformation descriptors:

```
<transformdb>
  <database key="string" value="::BigThree">
    <record>
      <if test="oldvalue == ::Old::Chrysler">
        <set target="newvalue"
              value="::New::DaimlerChrysler" />
      </if>
    </record>
  </database>
</transformdb>
```

When executed, these descriptors convert occurrences of `Chrysler` in the old type system into `DaimlerChrysler` in the transformed database's new type system. Transformation descriptors are described in detail in Section 23.4.

23.4 Transformation Descriptors

This section describes the XML elements comprising the FreezeScript transformation descriptors.

23.4.1 Overview

A transformation descriptor file has a well-defined structure. The top-level descriptor in the file is `<transformdb>`. A `<database>` descriptor must be present within `<transformdb>` to define the key and value types used by the database. Inside `<database>`, the `<record>` descriptor triggers the transformation process. See Section 23.3.5 for an example that demonstrates the structure of a minimal descriptor file.

During transformation, type-specific actions are supported by the `<transform>` and `<init>` descriptors, both of which are children of `<transformdb>`. One `<transform>` descriptor and one `<init>` descriptor may be defined for each type in the new Slice definitions. Each time **transformdb** creates a new instance of a type, it executes the `<init>` descriptor for that type, if one is defined. Similarly, each time **transformdb** transforms an instance of an old type into a new type, the `<transform>` descriptor for the new type is executed.

The `<database>`, `<record>`, `<transform>`, and `<init>` descriptors may contain general-purpose action descriptors such as `<if>`, `<set>`, and `<echo>`. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions make use of the expression language described in Section 23.8.

23.4.2 Flow of Execution

The transformation descriptors are executed as described below.

- `<database>` is executed first. Each child descriptor of `<database>` is executed in the order of definition. If a `<record>` descriptor is present, database transformation occurs at that point. Any child descriptors of `<database>` that follow `<record>` are not executed until transformation completes.
- During transformation of each record, **transformdb** creates instances of the new key and value types, which includes the execution of the `<init>` descriptors for those types. Next, the old key and value are transformed into the new key and value, in the following manner:
 1. Locate the `<transform>` descriptor for the type.
 2. If no descriptor is found, or the descriptor exists and it does not preclude default transformation, then transform the data as described in Section 23.3.1.
 3. If the `<transform>` descriptor exists, execute it.
 4. Finally, execute the child descriptors of `<record>`.

See Section 23.4.4 for detailed information on the transformation descriptors.

23.4.3 Scopes

The `<database>` descriptor creates a global scope, allowing child descriptors of `<database>` to define symbols that are accessible in any descriptor¹. Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the `<transform>` descriptor creates

1. In order for a global symbol to be available to a `<transform>` or `<init>` descriptor, the symbol must be defined before the `<record>` descriptor is executed.

a local scope and defines the symbols `old` and `new` to represent a value in its old and new forms. Child descriptors of `<transform>` can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during transformation. This can be accomplished as shown below:

```
<transformdb>
    <database key="string" value="::Ice::Identity">
        <define name="categoryCount" type="int" value="0"/>
        <record/>
        <echo message="categoryCount = " value="categoryCount"/>
    </database>
    <transform type="::Ice::Identity">
        <if test="new.category == 'Accounting'">
            <set target="categoryCount" value="categoryCount + 1"/>
        </if>
    </transform>
</transformdb>
```

In this example, the `<define>` descriptor introduces the symbol `categoryCount` into the global scope, defining it as type `int` with an initial value of zero. Next, the `<record>` descriptor causes transformation to proceed. Each occurrence of the type `Ice::Identity` causes its `<transform>` descriptor to be executed, which examines the `category` member and increases `categoryCount` if necessary. Finally, after transformation completes, the `<echo>` descriptor displays the final value of `categoryCount`.

To reinforce the relationships between descriptors and scopes, consider the diagram in Figure 23.1. Several descriptors are shown, including the symbols they define in their local scopes. In this example, the `<iterate>` descriptor has a dictionary target and therefore the default symbol for the element value, `value`, hides the symbol of the same name in the parent `<init>` descriptor's scope². In

2. This situation can be avoided by assigning a different symbol name to the element value.

addition to symbols in the `<iterate>` scope, child descriptors of `<iterate>` can also refer to symbols from the `<init>` and `<database>` scopes.

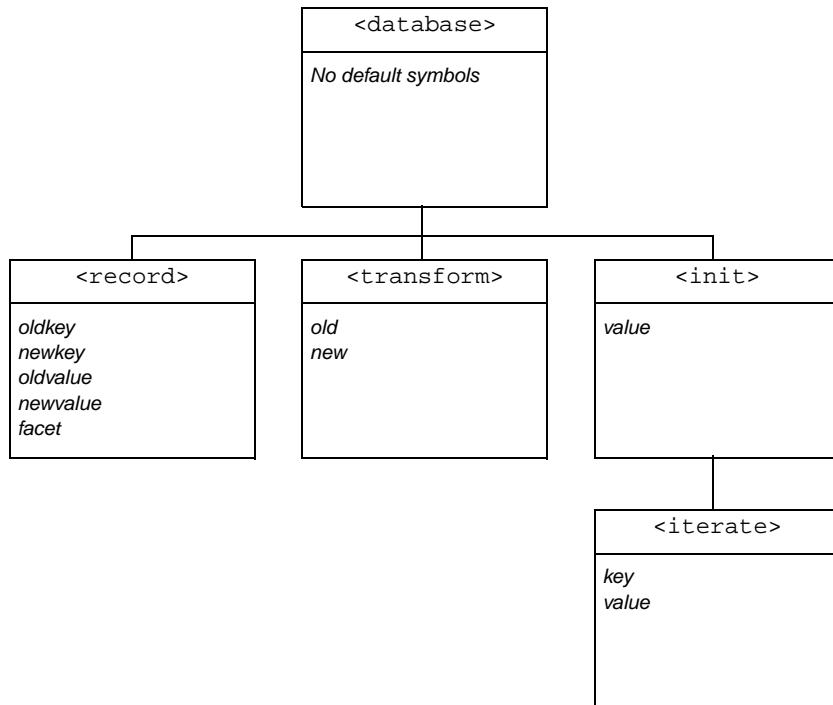


Figure 23.1. Relationship between descriptors and scopes.

23.4.4 Descriptor Reference

`<transformdb>`

The top-level descriptor in a descriptor file. It requires one child descriptor, `<database>`, and supports any number of `<transform>` and `<init>` descriptors. This descriptor has no attributes.

`<database>`

The attributes of this descriptor define the old and new key and value types for the database to be transformed. It supports any number of child descriptors, but at

most one `<record>` descriptor. The `<database>` descriptor also creates a global scope for user-defined symbols (see Section 23.4.3).

The attributes supported by the `<database>` descriptor are described in Table 23.2.

Table 23.2. Attributes for `<database>` descriptor.

Name	Description
<code>key</code>	Specifies the Slice types of the old and new keys. If the types are the same, only one needs to be specified. Otherwise, the types are separated by a comma.
<code>value</code>	Specifies the Slice types of the old and new values. If the types are the same, only one needs to be specified. Otherwise, the types are separated by a comma.

As an example, consider the following `<database>` descriptor. In this case, the Freeze map to be transformed currently has key type `int` and value type `::Employee`, and is migrating to a key type of `string`:

```
<database key="int,string" value="::Employee">
```

`<record>`

Commences the transformation. Child descriptors are executed for each record in the database, providing the user with an opportunity to examine the record's old key and value, and optionally modify the new key and value. Default transformations, as well as `<transform>` and `<init>` descriptors, are executed before the child descriptors. The `<record>` descriptor introduces the following symbols into a local scope: `oldkey`, `newkey`, `oldvalue`, `newvalue`, `facet`. These symbols are accessible to child descriptors, but not to `<transform>` or `<init>` descriptors. The `oldkey` and `oldvalue` symbols are read-only. The `facet` symbol is a string indicating the facet name of the object in the current record, and is only relevant for Freeze evictor databases.

Use caution when modifying database keys to ensure that duplicate keys do not occur. If a duplicate database key is encountered, transformation fails immediately.

Note that database transformation only occurs if a `<record>` descriptor is present.

<transform>

Customizes the transformation for all instances of a type in the new Slice definitions. The children of this descriptor are executed after the optional default transformation has been performed, as described in Section 23.3.1. Only one **<transform>** descriptor can be specified for a type, but a **<transform>** descriptor is not required for every type. The symbols `old` and `new` are introduced into a local scope and represent the old and new values, respectively. The `old` symbol is read-only. The attributes supported by this descriptor are described in Table 23.3.

Table 23.3. Attributes for **<transform>** descriptor.

Name	Description
<code>type</code>	Specifies the Slice type id.
<code>default</code>	If <code>false</code> , no default transformation is performed on values of this type. If not specified, the default value is <code>true</code> .
<code>base</code>	This attribute determines whether <transform> descriptors of base class types are executed. If <code>true</code> , the <transform> descriptor of the immediate base class is invoked. If no descriptor is found for the immediate base class, the class hierarchy is searched until a descriptor is found. The execution of any base class descriptors occurs after execution of this descriptor's children. If not specified, the default value is <code>true</code> .
<code>rename</code>	Indicates that a type in the old Slice definitions has been renamed to the new type identified by the <code>type</code> attribute. The value of this attribute is the type id of the old Slice definition. Specifying this attribute relaxes the strict compatibility rules defined in Section 23.3.2 for <code>enum</code> , <code>struct</code> and <code>class</code> types.

Below is an example of a **<transform>** descriptor that initializes a new data member:

```
<transform type="::Product">
    <set target="new.salePrice"
        value="old.listPrice * old.discount"/>
</transform>
```

For class types, **transformdb** first attempts to locate a **<transform>** descriptor for the object's most-derived type. If no descriptor is found,

transformdb proceeds up the class hierarchy in an attempt to find a descriptor. The base object type, `Object`, is the root of every class hierarchy and is included in the search for descriptors. It is therefore possible to define a `<transform>` descriptor for type `Object`, which will be invoked for every class instance.

Note that `<transform>` descriptors are executed recursively. For example, consider the following Slice definitions:

```
struct Inner {
    int sum;
};

struct Outer {
    Inner i;
};
```

When `transformdb` is performing the default transformation on a value of type `Outer`, it recursively performs the default transformation on the `Inner` member, then executes the `<transform>` descriptor for `Inner`, and finally executes the `<transform>` descriptor for `Outer`. However, if default transformation is disabled for `Outer`, then no transformation is performed on the `Inner` member and therefore the `<transform>` descriptor for `Inner` is not executed.

`<init>`

Defines custom initialization rules for all instances of a type. Child descriptors are executed each time the type is instantiated. The typical use case for this descriptor is for types that have been introduced in the new Slice definitions and whose instances require default values different than what `transformdb` supplies. The symbol `value` is introduced into a local scope to represent the instance. The attributes supported by this descriptor are described in Table 23.4.

Table 23.4. Attributes for `<init>` descriptor.

Name	Description
<code>type</code>	Specifies the Slice type id.

Here is a simple example of an `<init>` descriptor:

```
<init type="::Player">
    <set target="value.currency" value="100" />
</init>
```

Note that, like `<transform>`, `<init>` descriptors are executed recursively. For example, if an `<init>` descriptor is defined for a `struct` type, the `<init>` descriptors of the `struct`'s members are executed before the `struct`'s descriptor.

`<iterate>`

Iterates over a dictionary or sequence, executing child descriptors for each element. The symbol names selected to represent the element information may conflict with existing symbols in the enclosing scope, in which case those outer symbols are not accessible to child descriptors. The attributes supported by this descriptor are described in Table 23.5.

Table 23.5. Attributes for `<iterate>` descriptor.

Name	Description
<code>target</code>	The sequence or dictionary.
<code>index</code>	The symbol name used for the sequence index. If not specified, the default symbol is <code>i</code> .
<code>element</code>	The symbol name used for the sequence element. If not specified, the default symbol is <code>elem</code> .
<code>key</code>	The symbol name used for the dictionary key. If not specified, the default symbol is <code>key</code> .
<code>value</code>	The symbol name used for the dictionary value. If not specified, the default symbol is <code>value</code> .

Shown below is an example of an `<iterate>` descriptor that sets the new data member `reviewSalary` to `true` if the employee's salary is greater than \$3000.

```
<iterate target="new.employeeMap" key="id" value="emp">
    <if test="emp.salary > 3000">
        <set target="emp.reviewSalary" value="true" />
    </if>
</iterate>
```

<if>

Conditionally executes child descriptors. The attributes supported by this descriptor are described in Table 23.6.

Table 23.6. Attributes for <if> descriptor.

Name	Description
test	A boolean expression.

See Section 23.8 for more information on the descriptor expression language.

<set>

Modifies a value. The `value` and `type` attributes are mutually exclusive. If `target` denotes a dictionary element, that element must already exist (i.e., `<set>` cannot be used to add an element to a dictionary). The attributes supported by this descriptor are described in Table 23.7.

Table 23.7. Attributes for <set> descriptor.

Name	Description
target	An expression that must select a modifiable value.
value	An expression that must evaluate to a value compatible with the target's type.
type	The Slice type id of a class to be instantiated. The class must be compatible with the target's type.
length	An integer expression representing the desired new length of a sequence. If the new length is less than the current size of the sequence, elements are removed from the end of the sequence. If the new length is greater than the current size, new elements are added to the end of the sequence. If <code>value</code> or <code>type</code> is also specified, it is used to initialize each new element.
convert	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

The `<set>` descriptor below modifies a member of a dictionary element:

```
<set target="new.parts['P105J3'].cost"
      value="new.parts['P105J3'].cost * 1.05"/>
```

This `<set>` descriptor adds an element to a sequence and initializes its value:

```
<set target="new.partsList" length="new.partsList.length + 1"
      value='P105J3' />
```

`<add>`

Adds a new element to a sequence or dictionary. It is legal to add an element while traversing the sequence or dictionary using `<iterate>`, however the traversal order after the addition is undefined. The `key` and `index` attributes are mutually exclusive, as are the `value` and `type` attributes. If neither `value` nor `type` is specified, the new element is initialized with a default value. The attributes supported by this descriptor are described in Table 23.8.

Table 23.8. Attributes for `<add>` descriptor.

Name	Description
<code>target</code>	An expression that must select a modifiable sequence or dictionary.
<code>key</code>	An expression that must evaluate to a value compatible with the target dictionary's key type.
<code>index</code>	An expression that must evaluate to an integer value representing the insertion position. The new element is inserted before <code>index</code> . The value must not exceed the length of the target sequence.
<code>value</code>	An expression that must evaluate to a value compatible with the target dictionary's value type, or the target sequence's element type.
<code>type</code>	The Slice type id of a class to be instantiated. The class must be compatible with the target dictionary's value type, or the target sequence's element type.
<code>convert</code>	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

Below is an example of an <add> descriptor that adds a new dictionary element and then initializes its member:

```
<add target="new.parts" key="`P105J4' "/>
<set target="new.parts[`P105J4'].cost" value="3.15" />
```

<define>

Defines a new symbol in the current scope. The attributes supported by this descriptor are described in Table 23.9.

Table 23.9. Attributes for <define> descriptor.

Name	Description
name	The name of the new symbol. An error occurs if the name matches an existing symbol in the current scope.
type	The name of the symbol's formal Slice type. For user-defined types, the name should be prefixed with ::Old or ::New to indicate the source of the type. The prefix can be omitted for primitive types.
value	An expression that must evaluate to a value compatible with the symbol's type.
convert	If true, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Execution fails immediately if a range error occurs. If not specified, the default value is false.

Below are two examples of the <define> descriptor. The first example defines the symbol `identity` to have type `Ice::Identity`, and proceeds to initialize its members using <set>:

```
<define name="identity" type="::New::Ice::Identity" />
<set target="identity.name" value="steve" />
<set target="identity.category" value="Admin" />
```

The second example uses the enumeration we first saw in Section 23.3.5 to define the symbol `manufacturer` and assign it a default value:

```
<define name="manufacturer" type="::New::BigThree"
       value="::New::DaimlerChrysler" />
```

<remove>

Removes an element from a sequence or dictionary. It is legal to remove an element while traversing a sequence or dictionary using **<iterate>**, however the traversal order after removal is undefined. The attributes supported by this descriptor are described in Table 23.10.

Table 23.10. Attributes for **<remove>** descriptor.

Name	Description
target	An expression that must select a modifiable sequence or dictionary.
key	An expression that must evaluate to a value compatible with the key type of the target dictionary.
index	An expression that must evaluate to an integer value representing the index of the sequence element to be removed.

<fail>

Causes transformation to fail immediately. If **test** is specified, transformation fails only if the expression evaluates to **true**. The attributes supported by this descriptor are described in Table 23.11.

Table 23.11. Attributes for **<fail>** descriptor.

Name	Description
message	A message to display upon transformation failure.
test	A boolean expression.

The following **<fail>** descriptor terminates the transformation if a range error is detected:

```
<fail message="range error occurred in ticket count!"  
      test="old.ticketCount > 32767"/>
```

<delete>

Causes transformation of the current database record to cease, and removes the record from the transformed database. This descriptor has no attributes.

<echo>

Displays values and informational messages. If no attributes are specified, only a newline is printed. The attributes supported by this descriptor are described in Table 23.12.

Table 23.12. Attributes for <echo> descriptor.

Name	Description
message	A message to display.
value	An expression. The value of the expression is displayed in a structured format.

Shown below is an <echo> descriptor that uses both message and value attributes:

```
<if test="old.ticketCount > 32767">
    <echo message="deleting record with invalid ticket count: "
          value="old.ticketCount"/>
    <delete/>
</if>
```

23.4.5 Descriptor Guidelines

There are three points at which you can intercept the transformation process: when transforming a record (<record>), when transforming an instance of a type (<transform>), and when creating an instance of a type (<init>).

In general, <record> is used when your modifications require access to both the key and value of the record. For example, if the database key is needed as a factor in an equation, or to identify an element in a dictionary, then <record> is the only descriptor in which this type of modification is possible. The <record> descriptor is also convenient to use when the number of changes to be made is small, and does not warrant the effort of writing separate <transform> or <init> descriptors.

The `<transform>` descriptor has a more limited scope than `<record>`. It is used when changes must potentially be made to all instances of a type (regardless of the context in which that type is used) and access to the old value is necessary. The `<transform>` descriptor does not have access to the database key and value, therefore decisions can only be made based on the old and new instances of the type in question.

Finally, the `<init>` descriptor is useful when access to the old instance is not required in order to properly initialize a type. In most cases, this activity could also be performed by a `<transform>` descriptor that simply ignored the old instance, so `<init>` may seem redundant. However, there is one situation where `<init>` is required: when it is necessary to initialize an instance of a type that is introduced by the new Slice definitions. Since there are no instances of this type in the current database, a `<transform>` descriptor for that type would never be executed.

23.5 Using transformdb

This section describes the invocation of the `transformdb` tool, and provides advice on how best to use it. The tool supports the standard command-line options common to all Slice processors listed in Section 4.18, with the exception of the include directory (`-I`) option. The options specific to `transformdb` are described in the subsections below.

23.5.1 General Options

The following options are used in both automatic and custom migration:

- `--old SLICE`
- `--new SLICE`

Loads the old or new Slice definitions contained in the file `SLICE`. These options may be specified multiple times if several files must be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice containing only `#include` statements for each of the Slice files to be loaded. No duplication is possible in this case if the included files use include guards correctly.

- **--include-old DIR**
--include-new DIR

Adds the directory **DIR** to the set of include paths for the old or new Slice definitions.

- **--key TYPE[,TYPE]**
--value TYPE[,TYPE]

Specifies the Slice type(s) of the database key and value. If the type does not change, then the type only needs to be specified once. Otherwise, the old type is specified first, followed by a comma and the new type. For example, the option **--key int,string** indicates that the database key is migrating from **int** to **string**. On the other hand, the option **--key int,int** indicates that the key type does not change, and could be given simply as **--key int**. Type changes are restricted to those allowed by the compatibility rules defined in Section 23.3.2, but custom migration provides additional flexibility.

- **-e**

Indicates that a Freeze evictor database is being migrated. This option is required whenever **transformdb** operates on a Freeze evictor database. As a convenience, this option automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the **--key** and **--value** options are not necessary. Specifically, the key type of a Freeze evictor database is **Ice::Identity**, and the value type is **Freeze::ObjectRecord**. The latter is defined in the Slice file **Freeze/EvictorStorage.ice**; however, this file does not need to be loaded into your old and new Slice definitions.

- **-i**

Requests that **transformdb** ignore type changes that violate the compatibility rules defined in Section 23.3.2. If this option is not specified, transformation fails immediately if such a violation occurs. With this option, a warning is displayed but transformation continues.

- **-p**

Requests that **transformdb** purge object instances whose type is no longer found in the new Slice definitions. See Section 23.5.5 for more information.

- **-c**

Use catastrophic recovery on the old BerkeleyDB database environment.

- **-w**

Suppress duplicate warnings during transformation.

23.5.2 Database Arguments

If `transformdb` is invoked to transform a database, it requires three arguments:

- **dbenv**
The pathname of the database environment directory.
- **db**
The name of the existing database file. `transformdb` never modifies this database.
- **newdbenv**

The pathname of the database environment directory to contain the transformed database. This directory must exist, and must not contain an existing database whose name matches the `db` argument.

Upon successful transformation, a database of the same name is created in the new environment directory.

23.5.3 Automatic Migration

No additional arguments are necessary to use automatic migration. The standard options from Section 4.18, the general options described in Section 23.5.1, and the database arguments from Section 23.5.2 are all you need. For example, consider the following command, which uses automatic migration to transform a database with a key type of `int` and value type of `string` into a database with the same key type and a value type of `long`:

```
$ transformdb --key int --value string,long dbhome data.db newdbhome
```

Note that we did not need to specify the `--old` or `--new` options because our key and value types are primitives. Upon successful completion, the file `newdbhome/data.db` contains our transformed database.

23.5.4 Custom Migration

Analysis

Custom migration is a two-step process: you first write the transformation descriptors, and then execute them to transform a database. To assist you in the process of creating a descriptor file, **transformdb** can generate a default set of transformation descriptors by comparing your old and new Slice definitions. This feature is called *analysis*, and is enabled by specifying the following option:

- **-o FILE**

Specifies the descriptor file **FILE** to be created during analysis.

No database arguments are required, because transformation does not occur when the **-o** option is specified. The generated file contains a **<transform>** descriptor for each type that appears in both old and new Slice definitions, and an **<init>** descriptor for types that appear only in the new Slice definitions. In most cases, these descriptors are empty. However, they may contain XML comments describing changes detected by **transformdb** that may require action on your part.

For example, let us revisit the enumeration we defined in Section 23.3.5:

```
enum BigThree { Ford, Chrysler, GeneralMotors };
```

This enumeration has evolved into the one shown below. In particular, the **Chrysler** enumerator has been renamed to reflect a corporate merger:

```
enum BigThree { Ford, DaimlerChrysler, GeneralMotors };
```

Next we run **transformdb** in analysis mode:

```
$ transformdb --old old/BigThree.ice --new new/BigThree.ice \
--key string --value ::BigThree -o transform.xml
```

The generated file **transform.xml** contains the following descriptor for the enumeration **BigThree**:

```
<transform type="::BigThree">
    <!-- NOTICE: enumerator `Chrysler' has been removed -->
</transform>
```

The comment indicates that enumerator **Chrysler** is no longer present in the new definition, reminding us that we need to add logic in this **<transform>** descriptor to change all occurrences of **Chrysler** to **DaimlerChrysler**.

The descriptor file generated by **transformdb** is well-formed and does not require any manual intervention prior to being executed. However, executing an unmodified descriptor file is simply the equivalent of automatic migration.

Transformation

After preparing a descriptor file, either by writing one completely yourself, or modifying one generated by the analysis phase, you are ready to transform a database. One additional option is provided for transformation:

- **-f FILE**

Specifies the descriptor file **FILE** to be executed during transformation.

It is not necessary to provide the **--key** or **--value** options during transformation, because the key and value types are already specified in the file's **<database>** descriptor.

Continuing our enumeration example from the analysis discussion above, assume we have modified **transform.xml** to convert the Chrysler enumerator, and are now ready to execute the transformation:

```
$ transformdb --old old/BigThree.ice --new new/BigThree.ice \
-f transform.xml dbhome bigthree.db newbigthree.db
```

Strategies

If it becomes necessary for you to transform a Freeze database, we generally recommend that you attempt to use automatic migration first, unless you already know that custom migration is necessary. Since transformation is a non-destructive process, there is no harm in attempting an automatic migration, and it is a good way to perform a sanity check on your **transformdb** arguments (for example, to ensure that all the necessary Slice files are being loaded), as well as on the database itself. If **transformdb** detects any incompatible type changes, it displays an error message for each incompatible change and terminates without doing any transformation. In this case, you may want to run **transformdb** again with the **-i** option, which ignores incompatible changes and causes transformation to proceed.

Pay careful attention to any warnings that **transformdb** emits, as these may indicate the need for using custom migration. For example, if we had attempted to transform the database containing the **BigThree** enumeration from previous sections using automatic migration, any occurrences of the **Chrysler** enumerator would display the following warning:

```
warning: unable to convert 'Chrysler' to ::BigThree
```

If custom migration appears to be necessary, use analysis to generate a default descriptor file, then review it for NOTICE comments and edit as necessary. Liberal use of the `<echo>` descriptor can be beneficial when testing your descriptor file, especially from within the `<record>` descriptor where you can display old and new keys and values.

23.5.5 Transforming Objects

The polymorphic nature of Slice classes can cause problems for database migration. As an example, the Slice parser can ensure that a set of Slice definitions loaded into `transformdb` is complete for all types but classes (and exceptions, but we ignore those because they are not persistent). `transformdb` cannot know that a database may contain instances of a subclass derived from one of the loaded classes, but itself is not loaded. Alternatively, the type of a class instance may have been renamed and cannot be found in the new Slice definitions.

By default, these situations result in immediate transformation failure.

However, the `-p` option is a (potentially drastic) way to handle these situations: if a class instance has no equivalent in the new Slice definitions and this option is specified, `transformdb` removes the instance any way it can. If the instance appears in a sequence or dictionary element, that element is removed. Otherwise, the database record containing the instance is deleted.

Now, the case of a class type being renamed is handled easily enough using custom migration and the `rename` attribute of the `<transform>` descriptor. However, there are legitimate cases where the destructive nature of the `-p` option can be useful. For example, if a class type has been removed and it is simply easier to start with a database that is guaranteed not to contain any instances of that type, then the `-p` option may simplify the broader migration effort.

This is another situation in which running an automatic migration first can help point out the trouble spots in a potential migration. Using the `-p` option, `transformdb` emits a warning about the missing class type and continues, rather than halting at the first occurrence, enabling you to discover whether you have forgotten to load some Slice definitions, or need to rename a type.

23.6 Database Inspection

The FreezeScript tool `dumpdb` is used to examine a Freeze database. Its simplest invocation displays every record of the database, but the tool also supports more

selective activities. In fact, **dumpdb** supports a scripted mode that shares many of the same XML descriptors as **transformdb** (see Section 23.4), enabling sophisticated filtering and reporting.

23.6.1 Descriptor Overview

A **dumpdb** descriptor file has a well-defined structure. The top-level descriptor in the file is `<dumpdb>`. A `<database>` descriptor must be present within `<dumpdb>` to define the key and value types used by the database. Inside `<database>`, the `<record>` descriptor triggers database traversal. Shown below is an example that demonstrates the structure of a minimal descriptor file:

```
<dumpdb>
  <database key="string" value="::Employee">
    <record>
      <echo message="Key: " value="key"/>
      <echo message="Value: " value="value"/>
    </record>
  </database>
</dumpdb>
```

During traversal, type-specific actions are supported by the `<dump>` descriptor, which is a child of `<dumpdb>`. One `<dump>` descriptor may be defined for each type in the new Slice definitions. Each time **dumpdb** encounters an instance of a type, the `<dump>` descriptor for the type is executed.

The `<database>`, `<record>`, and `<dump>` descriptors may contain general-purpose action descriptors such as `<if>` and `<echo>`. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions make use of the expression language described in Section 23.8.

Although **dumpdb** descriptors are not allowed to modify the database, they can still define local symbols for scripting purposes. Once a symbol is defined by the `<define>` descriptor, other descriptors such as `<set>`, `<add>`, and `<remove>` can be used to manipulate the symbol's value.

23.6.2 Flow of Execution

The descriptors are executed as described below.

- `<database>` is executed first. Each child descriptor of `<database>` is executed in the order of definition. If a `<record>` descriptor is present,

database traversal occurs at that point. Any child descriptors of `<database>` that follow `<record>` are not executed until traversal completes.

- For each record, `dumpdb` interprets the key and value, invoking `<dump>` descriptors for each type it encounters. For example, if the value type of the database is a `struct`, then `dumpdb` first attempts to invoke a `<dump>` descriptor for the struct type, and then recursively interprets the structure's members in the same fashion.

See Section 23.6.4 for detailed information on the `dumpdb` descriptors.

23.6.3 Scopes

The `<database>` descriptor creates a global scope, allowing child descriptors of `<database>` to define symbols that are accessible in any descriptor³. Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the `<dump>` descriptor creates a local scope and defines the symbol `value` to represent a value of the specified type. Child descriptors of `<dump>` can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during database traversal. This can be accomplished as shown below:

```
<dumpdb>
    <database key="string" value="::Ice::Identity">
        <define name="categoryCount" type="int" value="0"/>
        <record/>
        <echo message="categoryCount = " value="categoryCount"/>
    </database>
    <dump type="::Ice::Identity">
        <if test="new.category == 'Accounting'">
            <set target="categoryCount" value="categoryCount + 1"/>
        </if>
    </dump>
</dumpdb>
```

3. In order for a global symbol to be available to a `<dump>` descriptor, the symbol must be defined before the `<record>` descriptor is executed.

In this example, the `<define>` descriptor introduces the symbol `categoryCount` into the global scope, defining it as type `int` with an initial value of zero. Next, the `<record>` descriptor causes traversal to proceed. Each occurrence of the type `Ice::Identity` causes its `<dump>` descriptor to be executed, which examines the `category` member and increases `categoryCount` if necessary. Finally, after traversal completes, the `<echo>` descriptor displays the final value of `categoryCount`.

To reinforce the relationships between descriptors and scopes, consider the diagram in Figure 23.2. Several descriptors are shown, including the symbols they define in their local scopes. In this example, the `<iterate>` descriptor has a dictionary target and therefore the default symbol for the element value, `value`, hides the symbol of the same name in the parent `<dump>` descriptor's scope⁴. In addition to symbols in the `<iterate>` scope, child descriptors of `<iterate>` can also refer to symbols from the `<dump>` and `<database>` scopes.

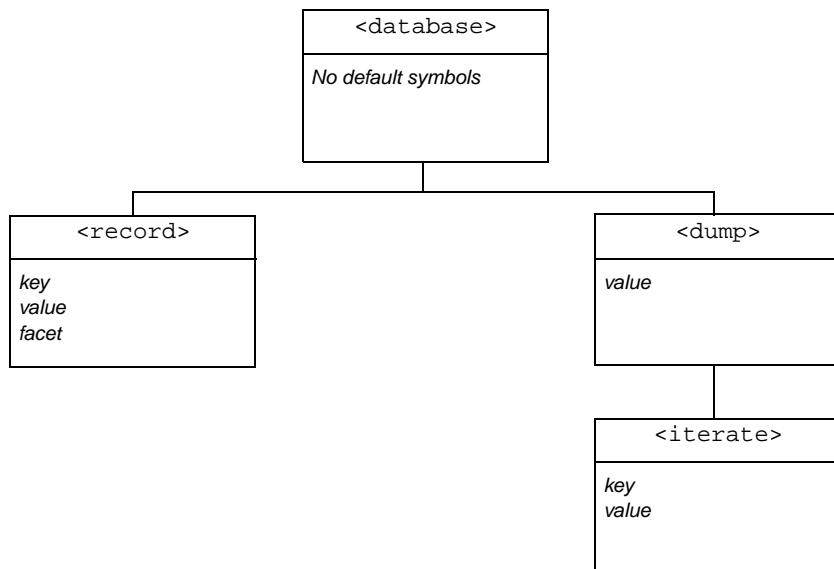


Figure 23.2. Relationship between descriptors and scopes.

4. This situation can be avoided by assigning a different symbol name to the element value.

23.6.4 Descriptor Reference

`<dumpdb>`

The top-level descriptor in a descriptor file. It requires one child descriptor, `<database>`, and supports any number of `<dump>` descriptors. This descriptor has no attributes.

`<database>`

The attributes of this descriptor define the key and value types of the database. It supports any number of child descriptors, but at most one `<record>` descriptor. The `<database>` descriptor also creates a global scope for user-defined symbols (see Section 23.6.3).

The attributes supported by the `<database>` descriptor are described in Table 23.13.

Table 23.13. Attributes for `<database>` descriptor.

Name	Description
<code>key</code>	Specifies the Slice type of the database key.
<code>value</code>	Specifies the Slice type of the database value.

As an example, consider the following `<database>` descriptor. In this case, the Freeze map to be examined has key type `int` and value type `::Employee`:

```
<database key="int" value="::Employee">
```

`<record>`

Commences the database traversal. Child descriptors are executed for each record in the database, but after any `<dump>` descriptors are executed. The `<record>` descriptor introduces the read-only symbols `key`, `value` and `facet` into a local scope. These symbols are accessible to child descriptors, but not to `<dump>` descriptors. The `facet` symbol is a string indicating the facet name of the object in the current record, and is only relevant for Freeze evictor databases.

Note that database traversal only occurs if a `<record>` descriptor is present.

<dump>

Executed for all instances of a Slice type. Only one <dump> descriptor can be specified for a type, but a <dump> descriptor is not required for every type. The read-only symbol `value` is introduced into a local scope. The attributes supported by this descriptor are described in Table 23.14.

Table 23.14. Attributes for <dump> descriptor.

Name	Description
<code>type</code>	Specifies the Slice type id.
<code>base</code>	If <code>type</code> denotes a Slice class, this attribute determines whether the <dump> descriptor of the base class is invoked. If <code>true</code> , the base class descriptor is invoked after executing the child descriptors. If not specified, the default value is <code>true</code> .
<code>contents</code>	For <code>class</code> and <code>struct</code> types, this attribute determines whether descriptors are executed for members of the value. For <code>sequence</code> and <code>dictionary</code> types, this attribute determines whether descriptors are executed for elements. If not specified, the default value is <code>true</code> .

Below is an example of a <dump> descriptor that searches for certain products:

```
<dump type="::Product">
    <if test="value.description.find('scanner') != -1">
        <echo message="Scanner SKU: " value="value.SKU"/>
    </if>
</dump>
```

For class types, `dumpdb` first attempts to locate a <dump> descriptor for the object's most-derived type. If no descriptor is found, `dumpdb` proceeds up the class hierarchy in an attempt to find a descriptor. The base object type, `Object`, is the root of every class hierarchy and is included in the search for descriptors. It is therefore possible to define a <dump> descriptor for type `Object`, which will be invoked for every class instance.

Note that <dump> descriptors are executed recursively. For example, consider the following Slice definitions:

```

struct Inner {
    int sum;
};

struct Outer {
    Inner i;
};

```

When **dumpdb** is interpreting a value of type Outer, it executes the <dump> descriptor for Outer, then recursively executes the <dump> descriptor for the Inner member, but only if the **contents** attribute of the Outer descriptor has the value **true**.

<iterate>

Iterates over a dictionary or sequence, executing child descriptors for each element. The symbol names selected to represent the element information may conflict with existing symbols in the enclosing scope, in which case those outer symbols are not accessible to child descriptors. The attributes supported by this descriptor are described in Table 23.15.

Table 23.15. Attributes for <iterate> descriptor.

Name	Description
target	The sequence or dictionary.
index	The symbol name used for the sequence index. If not specified, the default symbol is i .
element	The symbol name used for the sequence element. If not specified, the default symbol is elem .
key	The symbol name used for the dictionary key. If not specified, the default symbol is key .
value	The symbol name used for the dictionary value. If not specified, the default symbol is value .

Shown below is an example of an <iterate> descriptor that displays the name of an employee if the employee's salary is greater than \$3000.

```

<iterate target="value.employeeMap" key="id" value="emp">
    <if test="emp.salary > 3000">
        <echo message="Employee: " value="emp.name"/>
    </if>
</iterate>

```

<if>

Conditionally executes child descriptors. The attributes supported by this descriptor are described in Table 23.16.

Table 23.16. Attributes for <if> descriptor.

Name	Description
test	A boolean expression.

See Section 23.8 for more information on the descriptor expression language.

<set>

Modifies a value. The `value` and `type` attributes are mutually exclusive. If `target` denotes a dictionary element, that element must already exist (i.e., `<set>` cannot be used to add an element to a dictionary). The attributes supported by this descriptor are described in Table 23.17.

Table 23.17. Attributes for <set> descriptor.

Name	Description
target	An expression that must select a modifiable value.
value	An expression that must evaluate to a value compatible with the target's type.
type	The Slice type id of a class to be instantiated. The class must be compatible with the target's type.
length	An integer expression representing the desired new length of a sequence. If the new length is less than the current size of the sequence, elements are removed from the end of the sequence. If the new length is greater than the current size, new elements are added to the end of the sequence. If <code>value</code> or <code>type</code> is also specified, it is used to initialize each new element.
convert	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

The `<set>` descriptor below modifies a member of a dictionary element:

```
<set target="new.parts['P105J3'].cost"
      value="new.parts['P105J3'].cost * 1.05"/>
```

This `<set>` descriptor adds an element to a sequence and initializes its value:

```
<set target="new.partsList" length="new.partsList.length + 1"
      value='P105J3' />
```

`<add>`

Adds a new element to a sequence or dictionary. It is legal to add an element while traversing the sequence or dictionary using `<iterate>`, however the traversal order after the addition is undefined. The `key` and `index` attributes are mutually exclusive, as are the `value` and `type` attributes. If neither `value` nor `type` is specified, the new element is initialized with a default value. The attributes supported by this descriptor are described in Table 23.18.

Table 23.18. Attributes for `<add>` descriptor.

Name	Description
<code>target</code>	An expression that must select a modifiable sequence or dictionary.
<code>key</code>	An expression that must evaluate to a value compatible with the target dictionary's key type.
<code>index</code>	An expression that must evaluate to an integer value representing the insertion position. The new element is inserted before <code>index</code> . The value must not exceed the length of the target sequence.
<code>value</code>	An expression that must evaluate to a value compatible with the target dictionary's value type, or the target sequence's element type.
<code>type</code>	The Slice type id of a class to be instantiated. The class must be compatible with the target dictionary's value type, or the target sequence's element type.
<code>convert</code>	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

Below is an example of an `<add>` descriptor that adds a new dictionary element and then initializes its member:

```
<add target="new.parts" key="'P105J4' />
<set target="new.parts['P105J4'].cost" value="3.15" />
```

`<define>`

Defines a new symbol in the current scope. The attributes supported by this descriptor are described in Table 23.19.

Table 23.19. Attributes for `<define>` descriptor.

Name	Description
<code>name</code>	The name of the new symbol. An error occurs if the name matches an existing symbol in the current scope.
<code>type</code>	The name of the symbol's formal Slice type.
<code>value</code>	An expression that must evaluate to a value compatible with the symbol's type.
<code>convert</code>	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Execution fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

Below are two examples of the `<define>` descriptor. The first example defines the symbol `identity` to have type `Ice::Identity`, and proceeds to initialize its members using `<set>`:

```
<define name="identity" type="::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

The second example uses the enumeration we first saw in Section 23.3.5 to define the symbol `manufacturer` and assign it a default value:

```
<define name="manufacturer" type="::BigThree"
       value="::DaimlerChrysler"/>
```

`<remove>`

Removes an element from a sequence or dictionary. It is legal to remove an element while traversing a sequence or dictionary using `<iterate>`, however

the traversal order after removal is undefined. The attributes supported by this descriptor are described in Table 23.20.

Table 23.20. Attributes for <remove> descriptor.

Name	Description
target	An expression that must select a modifiable sequence or dictionary.
key	An expression that must evaluate to a value compatible with the key type of the target dictionary.
index	An expression that must evaluate to an integer value representing the index of the sequence element to be removed.

<fail>

Causes transformation to fail immediately. If `test` is specified, transformation fails only if the expression evaluates to `true`. The attributes supported by this descriptor are described in Table 23.21.

Table 23.21. Attributes for <fail> descriptor.

Name	Description
message	A message to display upon transformation failure.
test	A boolean expression.

The following <fail> descriptor terminates the transformation if a range error is detected:

```
<fail message="range error occurred in ticket count!"  
      test="value.ticketCount > 32767" />
```

<echo>

Displays values and informational messages. If no attributes are specified, only a newline is printed. The attributes supported by this descriptor are described in Table 23.22.

Table 23.22. Attributes for <echo> descriptor.

Name	Description
message	A message to display.
value	An expression. The value of the expression is displayed in a structured format.

Shown below is an <echo> descriptor that uses both message and value attributes:

```
<if test="value.ticketCount > 32767">
    <echo message="range error occurred in ticket count: "
          value="value.ticketCount"/>
</if>
```

23.7 Using `dumpdb`

This section describes the invocation of `dumpdb` and provides advice on how to best use it.

23.7.1 Options

The tool supports the standard command-line options common to all Slice processors listed in Section 4.18. The options specific to `dumpdb` are described below:

- **--load *SLICE***

Loads the Slice definitions contained in the file ***SLICE***. This option may be specified multiple times if several files must be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice containing

only `#include` statements for each of the Slice files to be loaded. No duplication is possible in this case if the included files use include guards correctly.

- **--key *TYPE***
--value *TYPE*
Specifies the Slice type of the database key and value.
- **-e**
Indicates that a Freeze evictor database is being examined. This option is required whenever `dumpdb` operates on a Freeze evictor database. As a convenience, this option automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the **--key** and **--value** options are not necessary. Specifically, the key type of a Freeze evictor database is `Ice::Identity`, and the value type is `Freeze::ObjectRecord`. The latter is defined in the Slice file `Freeze/EvictorStorage.ice`, however this file does not need to be explicitly loaded.
- **-o *FILE***
Create a file named ***FILE*** containing sample descriptors for the loaded Slice definitions. The key and value types must be specified using the **--key** and **--value** options, or using the **-e** option. If the **--select** option is used, its expression is included in the sample descriptors. The database arguments are not necessary because database traversal is not performed when the **-o** option is used.
- **-f *FILE***
Execute the descriptors in the file named ***FILE***. The file's <database> descriptor specifies the key and value types, therefore the **--key**, **--value** and **-e** options are not necessary when **-f** is used.
- **--select *EXPR***
Only display those records for which the expression ***EXPR*** is true. The expression can refer to the symbols `key` and `value`.

23.7.2 Database Arguments

If `dumpdb` is invoked to examine a database, it requires two arguments:

- **dbenv**

The pathname of the database environment directory.

- **db**

The name of the database file. `dumpdb` opens this database as read-only, and traversal occurs within a transaction.

23.7.3 Use Cases

The command line options described in Section 23.7.1 support several modes of operation:

- Dump an entire database.
- Dump selected records of a database.
- Emit a sample descriptor file.
- Execute a descriptor file.

These use cases are described in the following sections.

Dump an Entire Database

The simplest way to examine a database with `dumpdb` is to dump its entire contents. You must specify the database key and value types, load the necessary Slice definitions, and supply the names of the database environment directory and database file. For example, this command dumps a Freeze map database whose key type is `string` and value type is `Employee`:

```
$ dumpdb --key string --value ::Employee --load Employee.ice \
  db emp.db
```

Dump Selected Records

If only certain records are of interest to you, the `--select` option provides a convenient way to filter the output of `dumpdb`. In the following example, we select employees from the accounting department:

```
$ dumpdb --key string --value ::Employee --load Employee.ice \
  --select "value.dept == 'Accounting'" db emp.db
```

In cases where the database records contain polymorphic class instances, you must be careful to specify an expression that can be successfully evaluated against all records. For example, `dumpdb` fails immediately if, while evaluating the expression, a data member is encountered that does not exist in the class instance. The safest way to write an expression in this case is to check the type of the class instance before referring to any of its data members.

In the example below, we assume that a Freeze evictor database contains instances of various classes in a class hierarchy, and we are only interested in instances of Manager whose employee count is greater than 10:

```
$ dumpdb -e --load Employee.ice \
--select "value.servant.ice_id == '::Manager' and \
value.servant.group.length > 10" db emp.db
```

Alternatively, if Manager has derived classes, then the expression can be written in a different way so that instances of Manager and any of its derived classes are considered:

```
$ dumpdb -e --load Employee.ice \
--select "value.servant.ice_isA('::Manager') and \
value.servant.group.length > 10" db emp.db
```

Creating a Sample Descriptor File

If you require more sophisticated filtering or scripting capabilities, then you must use a descriptor file. The easiest way to get started with a descriptor file is to generate a template using **dumpdb**:

```
$ dumpdb --key string --value ::Employee --load Employee.ice \
-o dump.xml
```

The output file `dump.xml` is complete and can be executed immediately if desired, but typically the file is used as a starting point for further customization.

If the **--select** option is specified, its expression is included in the generated `<record>` descriptor as the value of the `test` attribute in an `<if>` descriptor.

Notice that database arguments are not required when **-o** is specified, because **dumpdb** terminates immediately after generating the sample file.

Executing a Descriptor File

Use the **-f** option when you are ready to execute a descriptor file. For example, we can execute the descriptor we generated in the previous section using this command:

```
$ dumpdb -f dump.xml --load Employee.ice db emp.db
```

23.8 Descriptor Expression Language

An expression language is provided for use in FreezeScript descriptors.

23.8.1 Operators

The language supports the usual complement of operators: `and`, `or`, `not`, `+`, `-`, `/`, `*`, `%`, `<`, `>`, `==`, `!=`, `<=`, `>=`, `(`, `)`. Note that the `<` character must be escaped as `<` in order to comply with XML syntax restrictions.

23.8.2 Literals

Literal values can be specified for integer, floating point, boolean, and string. The expression language supports the same syntax for literal values as that of Slice (see Section 4.7.5), with one exception: string literals must be enclosed in single quotes.

23.8.3 Symbols

Certain descriptors introduce symbols that can be used in expressions. These symbols must comply with the naming rules for Slice identifiers (i.e., a leading letter followed by zero or more alphanumeric characters). Data members are accessed using dotted notation, such as `value.memberA.memberB`.

Expressions can refer to Slice constants and enumerators using scoped names. In a `transformdb` descriptor, there are two sets of Slice definitions, therefore the expression must indicate which set of definitions it is accessing by prefixing the scoped name with `::Old` or `::New`. For example, the expression `old.fruitMember == ::Old::Pear` evaluates to `true` if the data member `fruitMember` has the enumerated value `Pear`. In `dumpdb`, only one set of Slice definitions is present and therefore the constant or enumerator can be identified without any special prefix.

23.8.4 Nil

The keyword `nil` represents a nil value of type `Object`. This keyword can be used in expressions to test for a nil object value, and can also be used to set an object value to nil.

23.8.5 Elements

Dictionary and sequence elements are accessed using array notation, such as `userMap['marc']` or `stringSeq[5]`. An error occurs if an expression attempts to access a dictionary or sequence element that does not exist. For dictio-

naries, the recommended practice is to check for the presence of a key before accessing it:

```
<if test="userMap.containsKey( 'marc' ) and userMap[ 'marc' ].active">
```

See Section 23.8.8 for more information on the `containsKey` function.

Similarly, expressions involving sequences should check the length of the sequence:

```
<if test="stringSeq.length > 5 and stringSeq[5] == 'fruit'">
```

See Section 23.8.7 for details on the `length` member.

23.8.6 Reserved Keywords

The following keywords are reserved: `and`, `or`, `not`, `true`, `false`, `nil`.

23.8.7 Implicit Members

Certain Slice types support implicit data members:

- Dictionary and sequence instances have a member `length` representing the number of elements.
- Object instances have a member `ice_id` denoting the actual type of the object.

23.8.8 Functions

The expression language supports two forms of function invocation: member functions and global functions. A member function is invoked on a particular data value, whereas global functions are not bound to a data value. For instance, here is an expression that invokes the `find` member function of a `string` value:

```
old.stringValue.find( 'theSubstring' ) != -1
```

And here is an example that invokes the global function `stringToIdentity`:

```
stringToIdentity( old.stringValue )
```

If a function takes multiple arguments, the arguments must be separated with commas.

String Member Functions

The `string` data type supports the following member functions:

- `int find(string match[, int start])`
Returns the index of the substring, or `-1` if not found. A starting position can optionally be supplied.
- `string replace(int start, int len, string str)`
Replaces a given portion of the string with a new substring, and returns the modified string.
- `string substr(int start[, int len])`
Returns a substring beginning at the given start position. If the optional length argument is supplied, the substring contains at most `len` characters, otherwise the substring contains the remainder of the string.

Dictionary Member Functions

The `dictionary` data type supports the following member function:

- `bool containsKey(key)`
Returns `true` if the dictionary contains an element with the given key, or `false` otherwise. The `key` argument must have a value that is compatible with the dictionary's key type.

Object Member Functions

Object instances support the following member function:

- `bool ice_isA(string id)`
Returns `true` if the object implements the given interface type, or `false` otherwise. This function cannot be invoked on a `nil` object.

Global Functions

The following global functions are provided:

- `string generateUUID()`
Returns a new UUID.
- `string identityToString(Ice::Identity id)`
Converts an identity into its string representation.
- `string lowercase(string str)`
Returns a new string converted to lowercase.
- `string proxyToString(Ice::ObjectPrx prx)`
Returns the string representation of the given proxy.

- `Ice::Identity stringToIdentity(string str)`
Converts a string into an `Ice::Identity`.
- `Ice::ObjectPrx stringToProxy(string str)`
Converts a string into a proxy.
- `string typeOf(val)`
Returns the formal Slice type of the argument.

23.9 Summary

FreezeScript provides tools that ease the maintenance of Freeze databases. The `transformdb` tool simplifies the task of migrating a database when its persistent types have changed, offering an automatic mode requiring no manual intervention, and a custom mode in which scripted changes are possible. Database inspection and reporting is accomplished using the `dumpdb` tool, which supports a number of operational modes including a scripting capability.

Chapter 24

IceSSL

24.1 Chapter Overview

In this chapter we present IceSSL, an optional security component for Ice applications. Section 24.2 provides an overview of the SSL protocol and the infrastructure required to support it. Section 24.3 and Section 24.4 discuss the configuration aspects of IceSSL, while Section 24.5 details the configuration file syntax.

Finally, Section 24.6 shows how an application can interact directly with IceSSL.

24.2 Introduction

Security is an important consideration for many distributed applications, both within corporate intranets as well as over untrusted networks, such as the Internet. The ability to protect sensitive information, ensure its integrity, and verify the identities of the communicating parties is essential for developing secure applications. With those goals in mind, Ice includes the IceSSL *plug-in* that provides these capabilities using the Secure Socket Layer (SSL) protocol.¹

1. IceSSL is currently only available for C++ applications.

24.2.1 SSL Overview

SSL is the protocol that enables Web browsers to conduct secure transactions and therefore is one of the most commonly used protocols for secure network communication. You do not need to know the technical details of the SSL protocol in order to use IceSSL successfully (and those details are outside the scope of this text). However, it would be helpful to have a high-level understanding of how the protocol works and the infrastructure required to support it. (For more information on the SSL protocol, see [22].)

SSL provides a secure environment for communication (without sacrificing too much performance) by combining a number of cryptographic techniques:

- public key encryption
- symmetric (shared key) encryption
- message authentication codes
- digital certificates

When a client establishes an SSL connection to a server, a *handshake* is performed. During a typical handshake, digital certificates that identify the communicating parties are validated, and symmetric keys are exchanged for encrypting the session traffic. Public key encryption, which is too slow to be used for the bulk of a session's data transfer, is used heavily during the handshaking phase. Once the handshake is complete, SSL uses message authentication codes to ensure data integrity, allowing the client and server to communicate at will with reasonable assurance that their messages are secure.

24.2.2 Public Key Infrastructure

Security requires trust, and public key cryptography by itself does nothing to establish trust. SSL addresses the issue of trust using Public Key Infrastructure (PKI), which binds public keys to identities using certificates. A Certification Authority (CA) is often used to issue certificates and verify the identities of certificate owners.

Smaller applications often require very little in the way of infrastructure; private certificates created for the client and server using freely-available tools may suffice. However, enterprises have more elaborate security requirements, in which setting up a private CA² or using a third-party CA (such as Verisign) is necessary.

It is also possible to avoid the use of certificates altogether; encryption is still used to obscure the session traffic, but the benefits of authentication are sacrificed in favor of reduced complexity and administration.

For more information on PKI, see [5].

24.2.3 Requirements

Integrating IceSSL into your application generally requires no changes to your source code, but does involve the following administrative tasks:

- creating a public key infrastructure (if necessary)
- writing the XML configuration files for the IceSSL plug-in
- modifying your application's configuration to install the IceSSL plug-in and use secure connections

The remainder of this chapter primarily discusses the configuration aspects of IceSSL.

24.3 Configuring IceSSL

IceSSL is configured separately for client and server functionality, reflecting the roles an Ice application can play: client only, server only, or mixed client-server. The configuration data is written in XML and provides the information IceSSL needs to initialize the SSL protocol, including the selection of cryptographic algorithms and the names of certificate and key files. This section presents example configuration files, while Section 24.5 provides a detailed reference of the XML elements comprising the IceSSL configuration.

24.3.1 RSA Example

Ice uses the configuration file shown below for its IceSSL-related tests and sample programs (see the file `certs/sslconfig.xml` in the Ice source distribution).

2. OpenSSL, the open-source SSL toolkit used by IceSSL, provides tools for setting up a minimal Certification Authority that may be suitable for development purposes or for a small organization.

The top-level element in every IceSSL configuration file is `SSLConfig`. Nested within this element is a `client` or `server` element (or in this case, both):

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE SSLConfig SYSTEM "sslconfig.dtd">
<SSLConfig>
    <client>
        <general version="SSLv23" cipherlist="RC4-MD5"
                 verifymode="peer" verifydepth="10" />
        <certauthority file="cacert.pem" />
        <basecerts>
            <rsacert keysize="1024">
                <public encoding="PEM"
                      filename="c_rsa1024_pub.pem" />
                <private encoding="PEM"
                      filename="c_rsa1024_priv.pem" />
            </rsacert>
        </basecerts>
    </client>
    <server>
        <general version="SSLv23" cipherlist="RC4-MD5"
                 verifymode="peer" verifydepth="10" />
        <certauthority file="cacert.pem" />
        <basecerts>
            <rsacert keysize="1024">
                <public encoding="PEM"
                      filename="s_rsa1024_pub.pem" />
                <private encoding="PEM"
                      filename="s_rsa1024_priv.pem" />
            </rsacert>
        </basecerts>
    </server>
</SSLConfig>
```

As you can see, the `client` and `server` elements are very similar. Let us examine the contents of `client` in more detail (the discussion applies equally well to the `server` element). The first element we encounter is `general`, which selects cryptographic algorithms and controls certificate verification:

```
<general version="SSLv23" cipherlist="RC4-MD5"
          verifymode="peer" verifydepth="10" />
```

The `version` attribute chooses a protocol version; in this case, the value `SSLv23` is really a compatibility mode that includes SSL version 2, SSL version

3, and Transport Layer Security (TLS) version 1. Note however that SSL version 2 has several security flaws and therefore is not supported by IceSSL.

A cipher suite is selected using the `cipherlist` attribute. The value RC4-MD5 chooses RC4 as the encryption algorithm and MD5 as the message digest algorithm.

The `verifymode` and `verifydepth` attributes affect certificate authentication. A verification mode of `peer` requires the client to authenticate the server's certificate before allowing the connection to proceed, and allows the client to send its certificate to the server upon request. A certificate may have an arbitrarily long chain of signing certificates that must be searched for a trusted CA; the `verifydepth` attribute allows you to establish a limit on the depth of the search. If this limit is exceeded, the connection fails.

Next, the `certauthority` element specifies the filename of a trusted CA certificate (or certificate chain):

```
<certauthority file="cacert.pem" />
```

As the certificate filename implies, it is encoded in the Privacy Enhanced Mail (PEM) format, which is the format IceSSL requires for all certificates and keys.

Finally, the `basecerts` element provides the public certificate and private key needed to identify the peer and authenticate certificates:

```
<basecerts>
    <rsacert keysize="1024">
        <public encoding="PEM"
            filename="c_rsa1024_pub.pem" />
        <private encoding="PEM"
            filename="c_rsa1024_priv.pem" />
    </rsacert>
</basecerts>
```

The `rsacert` element encapsulates the `public` and `private` elements and supplies the bit strength (1024) of the encryption key. The `public` element defines the filename and encoding of the public certificate. Similarly, the `private` element defines the filename and encoding of the private key. Although the encoding format is provided in an attribute, currently only PEM is supported.

24.3.2 ADH Example

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle

attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The IceSSL configuration file shown below demonstrates how to use ADH:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE SSLConfig SYSTEM "sslconfig.dtd">
<SSLConfig>
    <client>
        <general version="SSLv23" cipherlist="ADH" />
        <basecerts>
            <dhparams keysize="512" filename="dh512.pem" />
        </basecerts>
    </client>
    <server>
        <general version="SSLv23" cipherlist="ADH" />
        <basecerts>
            <dhparams keysize="512" filename="dh512.pem" />
        </basecerts>
    </server>
</SSLConfig>
```

The important aspects of this configuration are the selection of the ADH cipher and the use of a dhparams element to provide the Diffie-Hellman parameter file. We specify the same parameter file for both client and server, but they are not required to have the same parameters. During the handshake, the Diffie-Hellman parameters of one peer are shared with the other, so only one set of parameters is actually used. It is up to the SSL implementation to determine which peer's parameters are used, but usually it is the server's parameters. We do not specify any certificates, because there is no peer authentication.

24.3.3 Configuration File Strategies

The IceSSL configuration file provides a lot of flexibility. For example, if a client and a server have access to the same filesystem, keeping their IceSSL configurations in the same file can simplify administrative duties. However, it is also perfectly reasonable (and often necessary) to create separate files for each client and server, with the client's configuration file containing only a `client` element, and the server's configuration file containing only a `server` element.

If either the client or the server is a mixed-mode application (i.e., both sends and receives requests), then care must be taken when attempting to share IceSSL configurations: a mixed-mode application's configuration requires both `client`

and `server` elements, and it may not be appropriate for another peer to use the same configuration (keys, certificates, etc.).

24.4 Configuring Applications

Configuring an application to use IceSSL requires installing the plug-in and creating SSL endpoints.

24.4.1 Installing IceSSL

Ice supports a generic plug-in facility that allows extensions (such as IceSSL) to be installed dynamically without changing the application source code. The executable code for the IceSSL plug-in resides in a shared library on Unix and a dynamic link library (DLL) on Windows. The plug-in is installed using a configuration property:

```
Ice.Plugin.IceSSL=IceSSL:create
```

The last component of the property name (`IceSSL`) becomes the plug-in's official identifier for configuration purposes, but the IceSSL plug-in requires its identifier to be `IceSSL`. The property value `IceSSL:create` is sufficient to allow the Ice run time to locate the IceSSL library (on both Unix and Windows) and initialize the plug-in. The only requirement is that the library reside in a directory that appears in the library path (`LD_LIBRARY_PATH` on Unix, `PATH` on Windows). For more information on the `Ice.Plugin` property, see Appendix C.

The next step is to supply the plug-in with its configuration file(s), which is also accomplished via configuration properties:

```
IceSSL.Client.CertPath=/opt/certs  
IceSSL.Client.Config=sslconfig.xml  
IceSSL.Server.CertPath=/opt/certs  
IceSSL.Server.Config=sslconfig.xml
```

The properties are split into client and server versions, similar to the configuration data described in Section 24.3. A client need only specify the `IceSSL.Client` properties, and likewise for a server. When initializing the plug-in, the client configuration is read from the file specified by the `IceSSL.Client.Config` property, and the server configuration will be read from the file specified by the `IceSSL.Server.Config` property. (It is legal to use the same file for both

properties.) The `CertPath` properties specify a default directory in which the respective configuration and certificate files can be found.

The plug-in supports a number of additional configuration properties that are described in Appendix C, but the ones described above are sufficient for many applications.

24.4.2 Creating SSL Endpoints

Installing the IceSSL plug-in enables you to use a new protocol, `ssl`, in your endpoints. For example, the following endpoint list creates a TCP endpoint, an SSL endpoint, and a UDP endpoint:

```
MyAdapter.Endpoints=tcp -p 8000:ssl -p 8001:udp -p 8000
```

As this example demonstrates, it is possible for a UDP endpoint to use the same port number as a TCP or SSL endpoint, because UDP is a different protocol and therefore has its own set of ports. It is not possible for a TCP endpoint and an SSL endpoint to use the same port number, because SSL is essentially a layer over TCP.

Using SSL in stringified proxies is equally straightforward:

```
MyProxy=MyObject:tcp -p 8000:ssl -p 8001:udp -p 8000
```

For more information on proxies and endpoints, see Appendix D.

24.4.3 Security Considerations

Defining an object adapter's endpoints to use multiple protocols, as shown in the example in Section 24.4.2, has obvious security implications. If your intent is to use SSL to protect session traffic and/or restrict access to the server, then you should only define SSL endpoints.

There can be situations, however, in which insecure endpoint protocols are advantageous. Figure 24.1 illustrates an environment in which TCP endpoints are allowed behind the firewall, but external clients are required to use SSL.

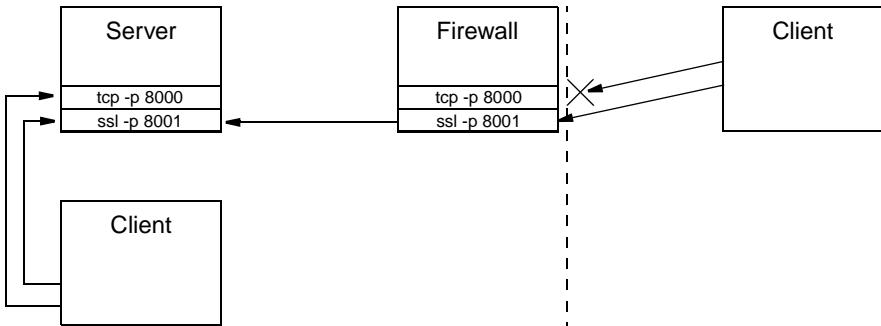


Figure 24.1. An application of multiple protocol endpoints.

The firewall in Figure 24.1 is configured to block external access to TCP port 8000 and to forward connections to port 8001 to the server machine.

One reason for using TCP behind the firewall is that it is more efficient than SSL and requires less administrative work to use. Of course, this scenario assumes that internal clients can be trusted, which is not true in many environments.

For more information on using SSL in complex network architectures, see Chapter 25.

24.5 Configuration Reference

This section describes each of the XML elements comprising the IceSSL configuration file.

24.5.1 Structure

The IceSSL configuration file has the following structure:

```
<SSLConfig>
  <client>
    <general .../>
    <certauthority .../>
    <basecerts>
      <rsacert ...>
```

```
        <public .../>
        <private .../>
    </rsacert>
    <dsacert ...>
        <public .../>
        <private .../>
    </dsacert>
    <dhpams .../>
</basecerts>
</client>
<server>
    <general .../>
    <certauthority .../>
    <basecerts>
        <rsacert ...>
            <public .../>
            <private .../>
        </rsacert>
        <dsacert ...>
            <public .../>
            <private .../>
        </dsacert>
        <dhpams .../>
    </basecerts>
    <tempcerts>
        <rsacert ...>
            <public .../>
            <private .../>
        </rsacert>
        <dhpams .../>
    </tempcerts>
</server>
</SSLConfig>
```

Some of the elements shown above are optional; see the element descriptions for details. A DTD for the configuration file is provided in the Ice source distribution as `certs/sslconfig.dtd`.

24.5.2 **basecerts**

The `basecerts` element contains certificate and key specifications. It is a mandatory child of `client` and `server`, and may contain one `rsacert` element, one `dsacert` element, and one `dhpams` element. In particular, `basecerts` must contain an `rsacert` element or a `dsacert` element (or

both). If a `dsacert` element is provided, a `dhparams` element normally accompanies it.

24.5.3 certauthority

The `certauthority` element specifies the file containing the chain of trusted CA certificates, as well as a directory in which certificates can be located. It is an optional child of `client` and `server`, and supports the following attributes:

Table 24.1. Attributes for certauthority element.

Attribute	Description	Required
file	A file (in PEM format) containing a chain of trusted CA certificates. A relative pathname is resolved using the directory defined by the <code>CertPath</code> property.	No
path	The absolute pathname of a directory containing trusted CA certificates. This directory needs to be prepared before use by running the OpenSSL utility <code>c_rehash</code> in the directory.	No

Both attributes are marked as optional, but at least one of the attributes should be defined. A significant difference between the attributes is the time at which the certificates are loaded. The certificate(s) indicated by the `file` attribute are loaded immediately, whereas the certificates residing in the directory indicated by the `path` attribute are loaded as necessary during the SSL handshake.

24.5.4 client

The `client` element establishes the configuration for the client-side components of the IceSSL plug-in. It is an optional child of `SSLConfig` and, if specified, it must contain a `general` element and a `basecerts` element, and may contain a `certauthority` element.

24.5.5 dhparams

The `dhparams` element specifies parameters for the Diffie-Hellman key agreement algorithm. It is an optional child of `basecerts` and `tempcerts`, and supports the following attributes:

Table 24.2. Attributes for `dhparams` element.

Attribute	Description	Required
<code>keysize</code>	The number of bits in the prime number parameter.	Yes
<code>encoding</code>	The encoding format of the file indicated by the <code>filename</code> attribute. The value must be <code>PEM</code> . If not specified, the default value is <code>PEM</code> .	No
<code>filename</code>	A file containing Diffie-Hellman parameters. A relative pathname is resolved using the directory defined by the <code>CertPath</code> property.	Yes

If this element is specified but the parameters cannot be read from the file specified by the `filename` attribute, IceSSL generates temporary Diffie-Hellman parameters using a 512-bit prime number.

24.5.6 dsacert

The `dsacert` element defines the public certificate and private key for the Digital Signature Algorithm (DSA). It is an optional child of `basecerts` and `tempcerts`, and must contain a `public` element and a `private` element. It supports the following attribute:

Table 24.3. Attributes for `dsacert` element.

Attribute	Description	Required
<code>keysize</code>	The bit strength of the keys.	Yes

24.5.7 general

The `general` element configures the SSL protocol. It is a mandatory child of `client` and `server`, and supports the following attributes:

Table 24.4. Attributes for general element.

Attribute	Description	Required
<code>version</code>	The SSL protocol version. Possible values are <code>SSLv23</code> , <code>SSLv3</code> , and <code>TLSv1</code> . The value <code>SSLv23</code> is a compatibility mode that includes SSL version 2, SSL version 3, and TLS version 1. Note that SSL version 2 has several security flaws and therefore is not supported by IceSSL. If not specified, the default value is <code>SSLv23</code> .	No
<code>cipherlist</code>	The list of cipher suites that SSL is allowed to use, separated by colons. If not specified, a default list of cipher suites is used.	No
<code>context</code>	A session ID context. Specifying this attribute enables session caching in a server.	No
<code>verifymode</code>	The certificate verification behavior. See Table 24.6 for the possible values and their semantics. Multiple values can be specified, separated by white space or a vertical bar (). If not specified, the default value is <code>none</code> .	No
<code>verifydepth</code>	The maximum depth when searching a certificate chain for a trusted CA. If this depth is exceeded, the handshake fails. If not specified, the default value is 10.	No
<code>randombytes</code>	Seeds the pseudo-random number generator with data from one or more files. A pathname may represent a Unix domain socket supporting the EGD protocol. Multiple pathnames must be separated by a semi-colon (;) on Windows and a colon (:) on Unix. At least 512 bytes of data must be provided.	No

cipherlist

The `cipherlist` attribute identifies the cipher suites that SSL is allowed to negotiate during the handshake. A cipher suite is a set of algorithms that satisfies the four requirements for establishing a secure connection: signing and authenti-

cation, key exchange, secure hashing, and encryption. Some algorithms satisfy more than one requirement, and there are many possible combinations.

The value of this attribute is given directly to the low-level OpenSSL library, on which IceSSL is based. Therefore, OpenSSL determines the allowable cipher suites, which in turn depend on how the OpenSSL distribution was compiled. You can obtain a complete list of the supported cipher suites using the **openssl** command:

```
$ openssl ciphers
```

This command will likely generate a long list. To simplify the selection process, OpenSSL supports several classes of ciphers:

Table 24.5. Cipher classes.

Class	Description
ALL	All possible combinations.
ADH	Anonymous ciphers.
LOW	Low bit-strength ciphers.
EXP	Export-crippled ciphers.

Classes and ciphers can be excluded by prefixing them with an exclamation point. The special keyword @STRENGTH sorts the cipher list in order of their strength, so that SSL gives preference to the more secure ciphers when negotiating a cipher suite. The @STRENGTH keyword must be the last element in the list.

For example, here is a good value for the `cipherlist` attribute:

```
ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH
```

This value excludes the ciphers with low bit strength and known problems, and orders the remaining ciphers according to their strength.

Note that no warning is given if an unrecognized cipher is specified.

verifymode

The `verifymode` attribute has the following semantics:

Table 24.6. Semantics of `verifymode` attribute.

Value	Client Semantics	Server Semantics
none	The client verifies the server's certificates, but failure does not terminate the handshake. The client does not send a certificate if one is requested by the server.	The server does not request a certificate from the client.
peer	The client verifies the server's certificates. Verification failure terminates the handshake.	The server requests a certificate from the client, and verifies it if one is sent. Verification failure terminates the handshake.
fail_no_cert	None.	Causes the handshake to fail if the client does not provide a certificate. Must be combined with <code>peer</code> .
client_once	None.	Prevents the server from requesting a certificate from the client during a renegotiation. Must be combined with <code>peer</code> .

The value `none` must not be combined with other values. The value `peer` must not be combined with other values in a client configuration.

24.5.8 private

The `private` element specifies the private key of a signing algorithm. It is a mandatory child of `rsacert` and `dsacert`, and supports the following attributes:

Table 24.7. Attributes for private element.

Attribute	Description	Required
<code>encoding</code>	The encoding format of the file indicated by the <code>filename</code> attribute. The value must be <code>PEM</code> . If not specified, the default value is <code>PEM</code> .	No
<code>filename</code>	A file containing the private key. A relative pathname is resolved using the directory defined by the <code>CertPath</code> property.	Yes

24.5.9 public

The `public` element specifies the public certificate of a signing algorithm. It is a mandatory child of `rsacert` and `dsacert`, and supports the following attributes:

Table 24.8. Attributes for public element.

Attribute	Description	Required
<code>encoding</code>	The encoding format of the file indicated by the <code>filename</code> attribute. The value must be <code>PEM</code> . If not specified, the default value is <code>PEM</code> .	No
<code>filename</code>	A file containing the public certificate. A relative pathname is resolved using the directory defined by the <code>CertPath</code> property.	Yes

24.5.10 rsacert

The `rsacert` element defines the RSA public certificate and private key. It is an optional child of `basecerts` and `tempcerts`, and must contain a `public` element and a `private` element. It supports the following attribute:

Table 24.9. Attributes for `dsacert` element.

Attribute	Description	Required
<code>keysize</code>	The bit strength of the keys.	Yes

24.5.11 server

The `server` element establishes the configuration for the server-side components of the IceSSL plug-in. It must contain a `general` element and a `basecerts` element, and may contain a `certauthority` element and a `tempcerts` element.

24.5.12 SSLConfig

The `SSLConfig` is the top-level element in an IceSSL configuration file. It must contain one `client` element, or one `server` element, or both.

24.5.13 tempcerts

The `tempcerts` element contains configuration parameters for temporary keys. It is an optional child of `server`, and may contain multiple `rsacert` and `dhpars` elements.

Temporary (also known as *ephemeral*) keys are required when using DSA authentication, and increase security when using RSA authentication (although ephemeral RSA keys are rarely used in practice). The `tempcerts` element allows you to specify files containing RSA public certificates and private keys, as well as Diffie-Hellman parameters. Loading these items from files avoids the need to generate them dynamically, which can be computationally expensive. A file is not read until a matching key size is requested by SSL; if no file is found with a matching key size, the requested data is generated dynamically.

24.6 Programming with IceSSL

Some applications may require IceSSL functionality that is only available programmatically. Here are a few reasons why an application might need direct interaction with the plug-in:

- To load configuration files dynamically
- To add keys and trusted certificates dynamically
- To install custom certificate verification rules

In order to accomplish any of these tasks, an application needs to obtain a reference to the plug-in from the communicator. Ice defines the Slice interface `Ice::Plugin` that represents a generic plug-in, and IceSSL defines a derived interface `IceSSL::Plugin`. These interfaces are described in Appendix B, but the following code example demonstrates how to obtain a reference to the IceSSL plug-in:

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr =
    communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin =
    IceSSL::PluginPtr::dynamicCast(plugin);
```

The first step is to obtain a reference to the plugin manager, then ask it for the IceSSL plugin. The argument to `getPlugin` is the plug-in identifier, in this case `IceSSL` (see Section 24.4.1). Finally, we downcast to `IceSSL::Plugin`.

24.7 Summary

The Secure Socket Layer (SSL) protocol is the de facto standard for secure network communication. Its support for authentication, non-repudiation, data integrity, and strong encryption makes it the logical choice for securing Ice applications.

Although security is an optional component of Ice, it is not an afterthought. The IceSSL plug-in integrates easily into existing Ice applications, in most cases requiring nothing more than configuration changes. Naturally, some additional effort is required to create the necessary security infrastructure for an application, but in many enterprises this work will have already been done.

Chapter 25

Glacier

25.1 Chapter Overview

This chapter presents Glacier, the firewall solution for Ice applications. Section 25.2 provides an introduction to Glacier and describes the complex networking issues Glacier is designed to address. Section 25.3 presents the simplest Glacier use case in which clients require only configuration changes. The issue of callbacks, and how Glacier supports callbacks in restricted network environments, is addressed in Section 25.4. The Glacier starter is the subject of Section 25.5, and Section 25.6 details the security aspects of using Glacier.

25.2 Introduction

We have presented many examples of client/server applications in this book, all of which assume the client and server programs are running either on the same host, or on multiple hosts with no network restrictions. We can justify this assumption because this is an instructional text, but a real-world network environment is usually much more complicated: client and server hosts with access to public networks often reside behind protective router-firewalls that not only restrict incoming connections, but also allow the protected networks to run in a private address space using Network Address Translation (NAT). These features, which

are practically mandatory in today’s hostile network environments, also disrupt the ideal world in which our examples are running.

25.2.1 Common Scenarios

Let us assume that a client and server need to communicate over an untrusted network, and that the client and server hosts reside in private networks behind firewalls, as shown in Figure 25.1.

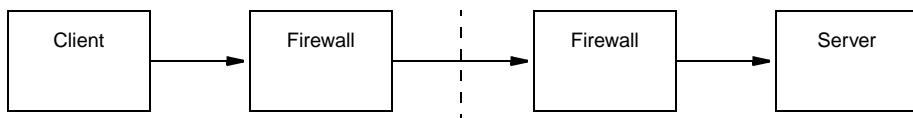


Figure 25.1. Client request in a typical network scenario.

Although the diagram looks fairly straightforward, there are several troublesome issues:

- A dedicated port on the server’s firewall must be opened and configured to forward messages to the server.
- If the server uses multiple endpoints (e.g., to support both TCP and SSL), then a firewall port must be dedicated to each endpoint.
- The client’s proxy must be configured to use the server’s “public” endpoint, which is the hostname and dedicated port of the firewall.
- If the server returns a proxy as the result of a request, the proxy must not contain the server’s private endpoint, which is inaccessible to the client.

To complicate the scenario even further, Figure 25.2 adds a callback from the server to the client. Callbacks imply that the client is also a server, therefore all of the issues associated with Figure 25.1 now apply to the client as well.

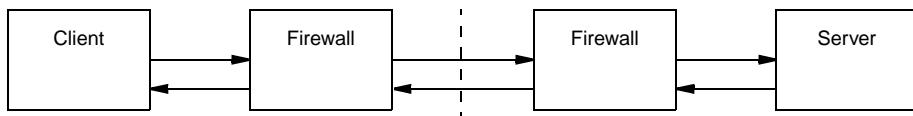


Figure 25.2. Callbacks in a typical network scenario.

As if this was not complicated enough already, Figure 25.3 adds multiple clients and servers. Each additional server (including clients requiring callbacks) adds more work for the firewall administrator as more ports are dedicated to forwarding requests.

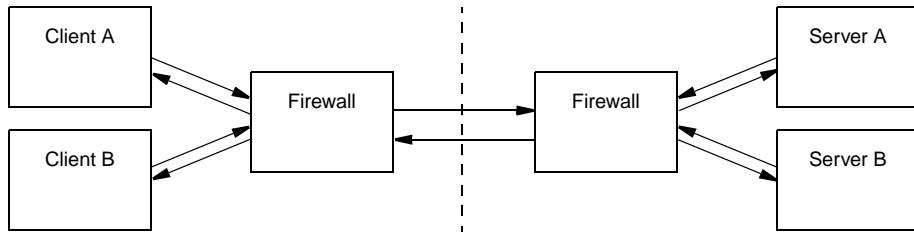


Figure 25.3. Multiple clients and servers with callbacks in a typical network scenario.

Clearly, these scenarios do not scale well, and are unnecessarily complex. Fortunately, Ice provides a solution.

25.2.2 What is Glacier?

Glacier, the router-firewall for Ice applications, addresses the issues raised in Section 25.2.1 with minimal impact on clients or servers (or firewall administrators). In Figure 25.4, Glacier becomes the server firewall for Ice applications. What is not obvious in the diagram, however, is how Glacier eliminates much of the complexity of the previous scenarios.

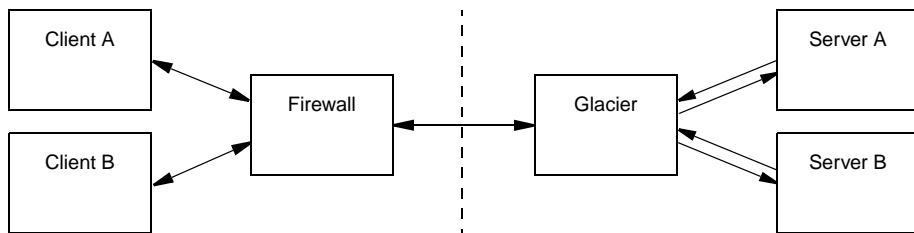


Figure 25.4. Multiple clients and servers with callbacks using Glacier.

In particular, Glacier provides the following advantages:

- Clients often require only configuration changes, not code changes, to use Glacier.
- A single Glacier port supports any number of servers.
- Servers are unaware of Glacier’s presence, and require no modifications whatsoever to use Glacier. From a server’s perspective, Glacier is just another local client, therefore servers are no longer required to advertise “public” endpoints in the proxies they create. Furthermore, back-end services such as IcePack (see Chapter 21) can continue to be used transparently behind the Glacier firewall.
- Callbacks are supported without requiring new connections from servers to clients. In other words, a callback from a server to a client is sent over an existing connection from the client to the server, thereby eliminating the administrative requirements associated with supporting callbacks in the client firewall.
- Glacier supports all Ice protocols (TCP, SSL, and UDP).
- Glacier requires no application-specific knowledge and therefore is very efficient: it routes request and reply messages without unmarshalling the message contents.

25.2.3 How it works

The Ice core supports a generic router facility, represented by the `Ice::Router` interface, that allows a third-party service to “intercept” requests on a properly-configured proxy and deliver them to the intended server. Glacier is an implementation of this service, although other implementations are certainly possible.

Glacier normally runs on a host with access to both networks: the public network over which clients send requests, and the private network over which those requests are routed to servers. Although it is possible in some cases to run Glacier behind a firewall, Glacier is really intended to *be* the firewall for Ice applications.

cations. It follows that Glacier must have endpoints on each network, as shown in Figure 25.5.

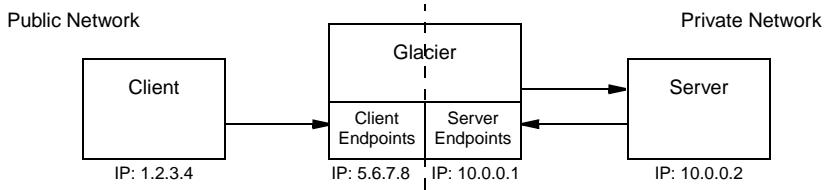


Figure 25.5. Glacier's client and server endpoints.

In the client, proxies must be configured to use Glacier as a router. This configuration can be done statically for all proxies created by a communicator, or programmatically for a particular proxy. A proxy configured to use a router is called a *routed proxy*.

When a client invokes an operation on a routed proxy, the client connects to one of Glacier's client endpoints and sends the request as if Glacier were the server. Glacier then establishes a client connection to the intended server, forwards the request, and returns the reply (if any). Glacier is essentially acting as a local client on behalf of the remote client.

If a server returns a proxy as the result of an operation, that proxy contains the server's endpoints in the private network, as usual. (Remember, the server is unaware of Glacier's presence, and therefore assumes that the proxy is usable by the client that requested it.) Of course, those endpoints are not accessible to the client and, in the absence of a router, the client would receive an exception if it were to use the proxy. When that proxy is configured with a router, however, the client ignores the server's endpoints and only sends requests to the router's client endpoints.

Glacier's server endpoints, which reside in the private network, are only used when a server makes a callback to a client. See Section 25.4 for more information on callbacks.

25.3 Using Glacier

The simplest Glacier configuration uses a single instance of the Glacier router. This configuration is capable of routing requests from multiple clients to multiple servers, but does not support callbacks from servers to clients.

Creating this configuration is straightforward:

1. Write a configuration file for the Glacier router.
2. Start the router on a host with access to the public and private networks.
3. Modify the client configuration to use the Glacier router.

For the sake of example, let us assume that the router's public address is 5.6.7.8 and its private address is 10.0.0.1.

25.3.1 Configuring the Router

The following router configuration properties establish the necessary endpoints:

```
Glacier.Router.Endpoints=tcp -h 5.6.7.8 -p 8000  
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8
```

The endpoint defined by `Glacier.Router.Endpoints` is called the *router control* endpoint because it is used by the Ice run time in a client to interact directly with the router. The endpoint defined by the property

`Glacier.Router.Client.Endpoints` is where requests from routed proxies are sent. Both of these endpoints must be accessible to clients and therefore are defined on the public network interface.¹

The router control endpoint uses a fixed port because clients may be statically configured with a proxy for this endpoint. The client endpoint, on the other hand, does not require a fixed port.

25.3.2 Starting the Router

Assuming the configuration properties shown in Section 25.3.1 are stored in a file named `config`, the router can be started with the following command:

```
$ glacierrouter --Ice.Config=config
```

25.3.3 Configuring the Client

The following property configures all of the proxies in a client to use a Glacier router:

1. This sample configuration uses TCP as the endpoint protocol, although in most cases SSL is preferable (see Section 25.6). One case where a TCP endpoint is necessary is for Java clients, since SSL is not currently supported in Ice for Java.

```
Ice.Default.Router=Glacier/router:tcp -h 5.6.7.8 -p 8000
```

The value of this property is a proxy for the router control object, therefore the endpoint must match the one in `Glacier.Router.Endpoints`.

25.3.4 Example

The `demo/Ice/Hello` example can be used to test this configuration. In fact, the example's configuration file already contains the relevant property definitions for using a Glacier router, but the file may require editing to enable some properties that are commented out by default. See the comments in the configuration file for more information.

25.4 Callbacks

Callbacks from servers to clients are commonly used in distributed applications, often for notification purposes (such as the completion of a long-running calculation or a change to a database record). Unfortunately, supporting callbacks in a complicated network environment presents its own set of problems, as described in Section 25.2.1. Ice overcomes these obstacles using a Glacier router and bi-directional connections.

25.4.1 Bi-directional Connections

While a regular unrouted connection only allows requests to flow in one direction (from client to server), a bi-directional connection enables requests to flow in both directions. This capability is necessary to circumvent the network restrictions discussed in Section 25.2.1, namely, client-side firewalls that prevent a server from establishing an independent connection directly to the client. By sending callback requests over the existing connection from the client to the server (more accurately, from the client to the router), we have created a virtual connection

back to the client. Figure 25.6 illustrates the steps involved in making a callback using Glacier.

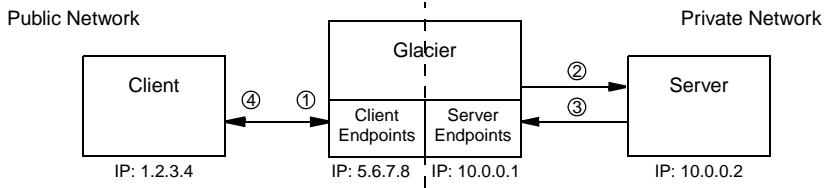


Figure 25.6. A callback via Glacier.

1. The client has a routed proxy for the server and makes an invocation. A connection is established to the router's client endpoint and the request is sent to the router.
2. The router, using information from the client's proxy, establishes a connection to the server and forwards the request. In this example, one of the arguments in the request is a proxy for a callback object in the client.
3. The server makes a callback to the client. For this to succeed, the proxy for the callback object must contain endpoints that are accessible to the server. The only path back to the client is through the router, therefore the proxy contains the router's server endpoints (see Section 25.4.4). The server connects to the router and sends the request.
4. The router forwards the callback request to the client using the bi-directional connection established in step 1.

The arrows in Figure 25.6 indicate the flow of requests; notice that two connections are used between the router and the server. Since the server is unaware of the router, it does not use routed proxies, and therefore does not use bi-directional connections.

25.4.2 Lifetime of a Bi-directional Connection

When a client terminates, it closes its connection to the router. If a server later attempts to make a callback to the client, the attempt fails because the router has no connection to the client over which to forward the request. This situation is no worse than if the server attempted to contact the client directly, which would be prevented by the client firewall. However, this illustrates the inherent limitation of

bi-directional connections: a client is only available for callbacks as long as it is connected to the router.

25.4.3 Callback Requirements

The principal issue we face in using callbacks is that a single instance of a Glacier router cannot forward callback requests to more than one client. Therefore, any client that uses callbacks must use its own instance of a Glacier router. This requirement presents an administrative dilemma: how do you (easily) ensure that each client gets its own router instance? We present the solution for this problem in Section 25.5; for now let us assume that we have only one client, and one router instance.

25.4.4 Configuring the Router

In order for the router to support callbacks from servers, it needs to have endpoints in the private network. The configuration file shown below adds the property `Glacier.Router.Server.Endpoints`:

```
Glacier.Router.Endpoints=tcp -h 5.6.7.8 -p 8000  
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8  
Glacier.Router.Server.Endpoints=tcp -h 10.0.0.1
```

Similar to the client endpoints described in Section 25.3.1, the server endpoints do not require fixed ports.

25.4.5 Configuring the Client Object Adapter

A client that receives callbacks is also a server, and therefore must have an object adapter. Typically, an object adapter has endpoints in the local network, but those endpoints are of no use to a server in our restricted network environment. We really want the proxy to use the router's server endpoints, and we accomplish that by configuring the object adapter with a proxy for the router. A single configuration property is all that we need:

```
Ice.Default.Router=Glacier/router:tcp -h 5.6.7.8 -p 8000
```

If you recall from Section 25.3.3, the `Ice.Default.Router` property configures all proxies with a router. In a server context, this property also configures all object adapters with a router. Note, however, that multiple object adapters created by the same communicator cannot use the same router, therefore we have to be

careful when using this property in a server context. Since our client only has one object adapter, this does exactly what we need (Section 25.4.7 discusses strategies for clients that require multiple object adapters).

For each object adapter, the Ice run time maintains a list of endpoints that are embedded in proxies created by that adapter. Normally this list simply contains the local endpoints defined for the object adapter but, when the adapter is configured with a router, the list also contains the router's server endpoints. In our callback example, the client does not need any local endpoints because it does not expect callbacks from local servers, so the proxies only contain the router's server endpoints.

It may seem unusual to create an object adapter that has no endpoints of its own and, in the absence of a router, there is no reason to do so. When using a router, however, the object adapter is necessary in order to service callback requests. Although it does not have local endpoints and therefore cannot receive new local connections, the object adapter does receive a “virtual” connection when the client establishes an outgoing connection to the router.

25.4.6 Active Connection Management

Ice supports *active connection management* (ACM). ACM conserves resources by periodically closing idle connections. This is a very useful feature, but generally should not be used for bi-directional connections, otherwise ACM might close a connection over which a client is still expecting a callback. ACM is enabled by default (see Appendix C for a description of the `Ice.ConnectionIdleTime` property).

25.4.7 Object Adapter Strategies

An application that needs to support callback requests from a router as well as requests from local clients should use multiple object adapters. This strategy ensures that proxies created by these object adapters contain the appropriate endpoints. For example, suppose we have the network configuration shown in

Figure 25.7. Notice that the two local area networks use the same private network addresses, which is not an unrealistic scenario.

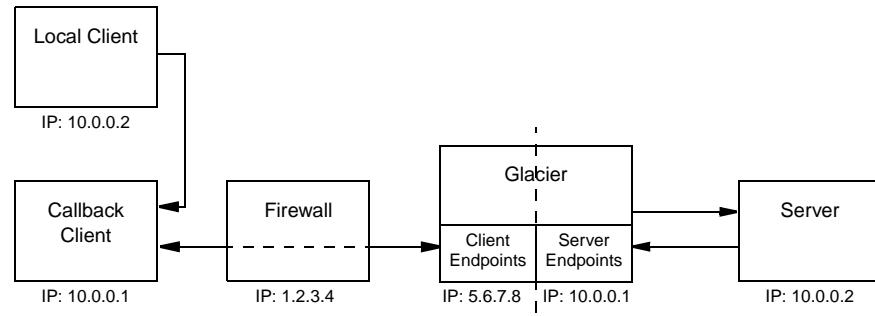


Figure 25.7. Supporting callback and local requests.

Now, if the callback client were to use a single object adapter for handling both callback requests and local requests, then any proxies created by that object adapter would contain the application's local endpoints as well as the router's server endpoints. As you might imagine, this could cause some subtle problems.

1. When the local client attempts to establish a connection to the callback client via one of these proxies, it might arbitrarily select one of the router's server endpoints to try first. Since the router's server endpoints use addresses in the same network, the local client attempts to make a connection over the local network, with two possible outcomes: the connection attempts to those endpoints fail, in which case they are skipped and the real local endpoints are attempted; or, even worse, one of the endpoints might accidentally be valid in the local network, in which case the local client has just connected to some unknown server.
2. The server may encounter similar problems when attempting to establish a local connection to the router in order to make a callback request.

The solution is to dedicate an object adapter solely to handling callback requests, and another one for servicing local clients.

In order to use multiple object adapters in this fashion, you must not define the `Ice.Default.Router` property (see Section 25.4.5). Assuming our callback object adapter is named `CallbackAdapter`, we configure the adapter with a router using the following property:

```
CallbackAdapter.Router=Glacier/router -h 5.6.7.8 -p 8000
```

Of course, by removing the `Ice.Default.Router` property, our client proxies are no longer automatically configured with a router. There are a couple of ways to solve this:

1. Invoke the `ice_router` operation on each proxy we need routed.
2. Use multiple communicators. For example, create one communicator that is configured with the `Ice.Default.Router` proxy, and use that communicator to create the callback object adapter and all routed proxies. Then create another communicator for the local object adapter.

25.4.8 Using Multiple Routers

A client is not limited to using only one router at a time: the proxy operation `ice_router` allows a client to configure its routed proxies as necessary. With respect to callbacks, the Ice core is designed to allow a single object adapter to accept callbacks from multiple routers, but that capability is not currently supported in the Glacier implementation. Therefore, a callback object adapter must be created for each router that can forward callback requests to the application.

25.4.9 Example

The `demo/Ice/callback` example can be used to test callbacks via Glacier. The example's configuration file already contains the relevant property definitions for using a Glacier router, but the file may require editing to enable some properties that are commented out by default. See the comments in the configuration file for more information.

25.5 The Glacier Starter

A Glacier router is capable of routing requests from multiple clients to multiple servers, but only when no callbacks are required (see Section 25.3). If callbacks

are necessary, each client must use its own Glacier router. Ice provides² the Glacier “starter” to launch routers on demand, as shown in Figure 25.8.

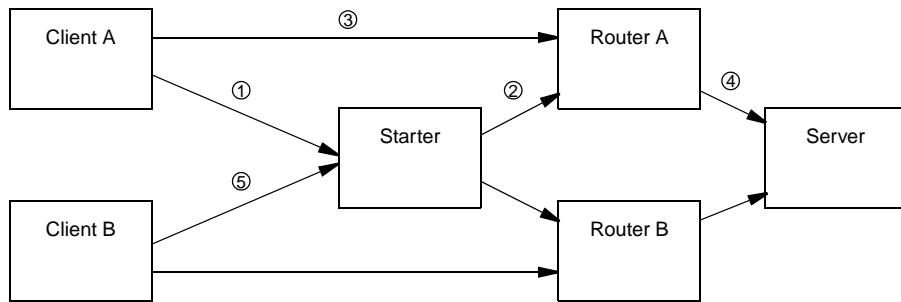


Figure 25.8. Using the Glacier starter.

1. Client A issues a request to the starter, which is listening on a well-known endpoint.
2. The starter spawns a new instance of the router, Router A, and returns the router’s proxy to the client.
3. Client A configures its proxies with the router, and makes an invocation via a routed proxy. From this point on, Client A interacts only with Router A, not the starter.
4. Router A forwards the request to the server.
5. The process is repeated for Client B.

As you can see, the Glacier starter is a very convenient way of solving the router-per-client issue. For security considerations, see Section 25.6.

25.5.1 Configuring the Starter

The starter is a regular Ice server and supports a number of configuration properties, as described in Appendix C. A minimal configuration is shown below:

2. The Glacier starter is currently only provided for Unix platforms.

```
Glacier.Starter.Endpoints=tcp -h 5.6.7.8 -p 8000
Glacier.Starter.RouterPath=/opt/Ice/bin/glacierrouter
Glacier.Starter.PropertiesOverride=Ice.ServerIdleTime=60

Glacier.Router.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Server.Endpoints=tcp -h 10.0.0.1
```

The `Glacier.Starter.Endpoints` property defines the well-known endpoint for the starter. This is the endpoint that a client uses in its starter proxy (see Section 25.5.3).

The `Glacier.Starter.RouterPath` property supplies the pathname of the router executable. If this property is not defined, the default value is `glacierrouter`, implying that the router executable must reside in the starter's executable search path.

The most interesting property is `Glacier.Starter.Properties-Override`. A router that is launched by a starter inherits the starter's configuration properties, which is why we included the router's properties in the starter's configuration. However, you may want to override one of the starter's properties, or you may want to augment the properties specifically for the router without applying them to the starter. The `PropertiesOverride` property allows you to do this.

In this example, the starter defines the `Ice.ServerIdleTime` property, which forces a server to terminate gracefully if there is no connection activity for a certain period of time. We do not set this property on the starter because we want it to remain available for new client requests. However, this is a very useful property for the router because it avoids the accumulation of inactive router processes whose clients have terminated or crashed. We therefore use the `Properties-Override` property to define the `Ice.ServerIdleTime` property only for the router.

Finally, we define the router's endpoints. These endpoints do not use fixed ports because each router is launched for use by a single client and therefore must use system-assigned ports. If we were to specify fixed ports, then only one router at a time would be able to run.

25.5.2 Starting the Starter

Assuming the configuration properties shown in Section 25.5.1 are stored in a file named `config`, the starter can be started with the following command:

```
$ glacierstarter --Ice.Config=config
```

The starter can also be run as a Unix daemon or Win32 service. See Section 10.3.2 for more information.

25.5.3 Using the Starter

A client can obtain a router by invoking the `startRouter` operation on the `Glacier::Starter` interface.

C++ Example

The example code below illustrates how a C++ client obtains a router.

```
#include <Ice/Ice.h>
#include <Glacier/Glacier.h>

// ...

Ice::CommunicatorPtr communicator = // ...
Ice::ObjectAdapterPtr adapter = // ...

std::string ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice::ObjectPrx obj = communicator->stringToProxy(ref);
Glacier::StarterPrx starter =
    Glacier::StarterPrx::checkedCast(obj);

Ice::ByteSeq privateKey, publicKey, routerCert;

Glacier::RouterPrx router =
    starter->startRouter("user", "passwd", privateKey, publicKey,
                          routerCert);

communicator->setDefaultRouter(router);
adapter->addRouter(router);
```

The code begins by creating a proxy for the starter. The endpoint in this proxy matches the one we configured using the `Glacier.Starter.Endpoints` property in Section 25.5.1.

```
std::string ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice::ObjectPrx obj = communicator->stringToProxy(ref);
Glacier::StarterPrx starter =
    Glacier::StarterPrx::checkedCast(obj);
```

Next, we invoke the `startRouter` operation, passing values for the user and password arguments. (See Section 25.6 for more information on starter security.)

```
Ice::ByteSeq privateKey, publicKey, routerCert;

Glacier::RouterPrx router =
    starter->startRouter("user", "passwd", privateKey, publicKey,
                          routerCert);
```

Finally, we configure the communicator and the object adapter with the router proxy we received from the starter.

```
communicator->setDefaultRouter(router);
adapter->addRouter(router);
```

At this point, the communicator is ready to create routed proxies, and the object adapter is ready to receive callback requests.

Java Example

The example code below illustrates how a Java client obtains a router.

```
Ice.Communicator communicator = // ...
Ice.ObjectAdapter adapter = // ...

String ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice.ObjectPrx obj = communicator.stringToProxy(ref);
Glacier.StarterPrx starter =
    Glacier.StarterPrxHelper.checkedCast(obj);

Ice.ByteSeqHolder privateKey = new Ice.ByteSeqHolder();
Ice.ByteSeqHolder publicKey = new Ice.ByteSeqHolder();
Ice.ByteSeqHolder routerCert = new Ice.ByteSeqHolder();

Glacier.RouterPrx router =
    starter.startRouter("user", "passwd", privateKey, publicKey,
                        routerCert);

communicator.setDefaultRouter(router);
adapter.addRouter(router);
```

The code begins by creating a proxy for the starter. The endpoint in this proxy matches the one we configured using the `Glacier.Starter.Endpoints` property in Section 25.5.1.

```
String ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice.ObjectPrx obj = communicator.stringToProxy(ref);
Glacier.StarterPrx starter =
    Glacier.StarterPrxHelper.checkedCast(obj);
```

Next, we invoke the `startRouter` operation, passing values for the user and password arguments. (See Section 25.6 for more information on starter security.)

```
Ice.ByteSeqHolder privateKey = new Ice.ByteSeqHolder();
Ice.ByteSeqHolder publicKey = new Ice.ByteSeqHolder();
Ice.ByteSeqHolder routerCert = new Ice.ByteSeqHolder();

Glacier.RouterPrx router =
    starter.startRouter("user", "passwd", privateKey, publicKey,
                        routerCert);
```

Finally, we configure the communicator and the object adapter with the router proxy we received from the starter.

```
communicator.setDefaultRouter(router);
adapter.addRouter(router);
```

At this point, the communicator is ready to create routed proxies, and the object adapter is ready to receive callback requests.

25.6 Glacier Security

As a firewall, a Glacier router represents a doorway into a private network, and in most cases that doorway should have a good lock. The obvious first step is to use SSL for the router control, router client, and starter endpoints. This allows you to secure the message traffic and restrict access to clients having the proper credentials (see Chapter 24). However, the Glacier starter takes security one step further by providing even stricter access control capabilities. We saw evidence of these capabilities in the sample code presented in Section 25.5.3, and this section explores them in depth.

25.6.1 Access Control

For some applications, the authentication capabilities of SSL are not sufficient to restrict access to the firewall. The certificate validation phase of the SSL hand-shake verifies that the user is who he says he is, but how do we know that he should be given passage through the firewall?

The Glacier starter addresses this issue through the use of an access control facility. The first two arguments to the `startRouter` operation are a username and password, as shown in Section 25.5.3. These arguments are verified by the starter

before it launches a router for the client. (It is safe for the client to send these arguments “in the clear” because the client connects to the starter via SSL.)

The starter supports two forms of username/password verification: a file-based access control list, or a custom verifier implementation. Configuration properties determine which form of verification a particular starter uses; file-based verification is used by default.

Crypt File

The file-based access control list uses the `crypt` algorithm to protect the passwords, similar to the `passwd` file on a typical Unix system. Each line of the file must contain a username and password, separated by white space. The password must be a 13-character, `crypt`-encoded string. For example, the file used by the test suite in `test/Glacier/starter/passwords` contains the following line:

```
dummy xxMqsnnDcK8tw
```

You can obtain the `crypt` encoding for a password using the `openssl` utility provided with OpenSSL:

```
$ openssl passwd
```

The program prompts you for a password and generates the `crypt`-encoded version for use in the password file.

The `Glacier.Starter.CryptPasswords` property defines the name of the file. If not specified, the default name is `passwords`.

Permissions Verifier

An application that has special requirements can implement the interface `Glacier::PermissionsVerifier` to have programmatic control over access to a starter. This can be especially useful in situations where a repository of account information already exists (such as an LDAP directory), in which case duplicating that information in another file would be tedious and error prone.

The Slice definition for the interface contains just one operation:

```
module Glacier {
interface PermissionsVerifier {
    nonmutating
    bool checkPermissions(string userId, string password,
                           out string reason);
};
```

The starter invokes `checkPermissions` on the verifier object, passing it the user-name and password arguments that were given to `startRouter`. The operation must return true if the arguments are valid, false otherwise. If the operation returns false, a reason can be provided in the output parameter.

To configure a starter with a custom verifier, set the configuration property `Glacier.Starter.PermissionsVerifier` with the proxy for the object. Since the username and password are sent in the clear, it is advisable to use an SSL endpoint for the verifier.

25.6.2 Filtering

The Glacier router is capable of filtering requests based on object identity, which helps to ensure that clients do not gain access to unintended objects.

As you may recall, the `Ice::Identity` type contains two string members: `category` and `name`. A router can be configured with a list of valid identity categories, in which case it only routes requests for objects in those categories. The configuration property `Glacier.Router.AllowCategories` supplies the category list:

```
Glacier.Router.AllowCategories=cat1 cat2
```

The starter also plays a role in this filtering capability. The configuration property `Glacier.Starter.AddUserToAllowCategories` controls whether the starter, when launching a router, adds the verified username to the router's list of valid categories. This feature accommodates applications in which Ice objects are created for use by a particular client and must not be accessible to other clients. Since the starter is creating a new router for each client, it is in the ideal position to supply a client-specific category (i.e., the username) to the router. This strategy naturally assumes the server is cooperating by using the username as the identity category of the Ice objects it creates for the client. The properties shown below demonstrate how to configure the starter and router for filtering:

```
Glacier.Starter.Endpoints=tcp -h 5.6.7.8 -p 8000
Glacier.Starter.RouterPath=/opt/Ice/bin/glacierrouter
Glacier.Starter.PropertiesOverride=Ice.ServerIdleTime=60
Glacier.Starter.AddUserToAllowCategories=1

Glacier.Router.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Server.Endpoints=tcp -h 10.0.0.1
Glacier.Router.AllowCategories=Factory
```

Using this configuration, a router created for user `duncanfoo` will allow requests for the identity categories `Factory` and `duncanfoo`. If that client makes a request for any other identity category, it receives an `Ice::ObjectNotExistException`.

25.6.3 Securing the Router

In addition to the access control and filtering features discussed in the previous sections, the Glacier starter takes one more step to secure the router. When the starter is configured for IceSSL and a new client is successfully verified, the starter generates a key pair for the client and a key pair for the router. The starter supplies the client's public key to the router, and the router's public key to the client. The router and client then configure their respective IceSSL plug-ins with their peer's public key, in order to ensure a connection is established only with that peer.

The code for properly configuring the IceSSL plug-in in a client is shown below. We include the steps for invoking `startRouter`, although they are identical to the examples presented in Section 25.5.3 and therefore are not described again. The goal of this example is to demonstrate how to use the three out arguments returned by `startRouter`.

```
#include <Ice/Ice.h>
#include <Glacier/Glacier.h>
#include <IceUtil/Base64.h>
#include <IceSSL/Plugin.h>

// ...

Ice::CommunicatorPtr communicator = // ...
Ice::ObjectAdapterPtr adapter = // ...

std::string ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice::ObjectPrx obj = communicator->stringToProxy(ref);
Glacier::StarterPrx starter =
    Glacier::StarterPrx::checkedCast(obj);

Ice::ByteSeq privateKey, publicKey, routerCert;

Glacier::RouterPrx router =
    starter->startRouter("user", "passwd", privateKey, publicKey,
                           routerCert);
```

```
Ice::PropertiesPtr properties = communicator->getProperties();

string cliCfg = properties->getProperty("IceSSL.Client.Config");
string srvCfg = properties->getProperty("IceSSL.Server.Config");

if (!cliCfg.empty() && !srvCfg.empty())
{
    string privateKeyBase64 = IceUtil::Base64::encode(privateKey);
    string publicKeyBase64 = IceUtil::Base64::encode(publicKey);
    string routerCertString = IceUtil::Base64::encode(routerCert);

    Ice::PluginManagerPtr pluginManager =
        communicator->getPluginManager();
    Ice::PluginPtr plugin = pluginManager->getPlugin("IceSSL");
    IceSSL::PluginPtr sslPlugin =
        IceSSL::PluginPtr::dynamicCast(plugin);

    sslPlugin->configure(IceSSL::Server);
    sslPlugin->setCertificateVerifier(
        IceSSL::ClientServer,
        sslPlugin->getSingleCertVerifier(routerCert));

    sslPlugin->setRSAKeysBase64(IceSSL::ClientServer,
                                 privateKeyBase64,
                                 publicKeyBase64);
    sslPlugin->addTrustedCertificateBase64(IceSSL::ClientServer,
                                             routerCertString);
}

communicator->setDefaultRouter(router);
adapter->addRouter(router);
```

After invoking `startRouter`, the client determines whether it is configured for IceSSL by checking for the IceSSL configuration properties. If both properties are defined, the client continues with the plug-in configuration.

The values returned by `startRouter` are converted to a Base64 encoding, which is the format required by IceSSL:

```
string privateKeyBase64 = IceUtil::Base64::encode(privateKey);
string publicKeyBase64 = IceUtil::Base64::encode(publicKey);
string routerCertString = IceUtil::Base64::encode(routerCert);
```

Next, the client obtains a reference to the IceSSL plug-in using the technique described in Chapter 24.

```
Ice::PluginManagerPtr pluginManager =
    communicator->getPluginManager();
Ice::PluginPtr plugin = pluginManager->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin =
    IceSSL::PluginPtr::dynamicCast(plugin);
```

The client then manually configures the server component of the IceSSL plug-in, which causes the plug-in to load the IceSSL server configuration file, and then creates a certificate verifier using the router's public key.

```
sslPlugin->configure(IceSSL::Server);
sslPlugin->setCertificateVerifier(
    IceSSL::ClientServer,
    sslPlugin->getSingleCertVerifier(routerCert));
```

Finally, the client gives its new RSA key pair to the plug-in, and then adds the router's public key as a trusted certificate.

```
sslPlugin->setRSAKeysBase64(IceSSL::ClientServer,
                             privateKeyBase64,
                             publicKeyBase64);
sslPlugin->addTrustedCertificateBase64(IceSSL::ClientServer,
                                         routerCertString);
```

25.7 Summary

Complex network environments are a fact of life. Unfortunately, the cost of securing an enterprise's network is increased application complexity and administrative overhead. Glacier helps to minimize these costs by providing a low-impact, efficient and secure router-firewall for Ice applications.

Chapter 26

IceBox

26.1 Chapter Overview

In this chapter we present IceBox, an easy-to-use framework for Ice application services. Section 26.2 provides an overview of IceBox and the advantages of using it. Section 26.3 introduces the service manager and describes its role in IceBox. A tutorial on writing and configuring an IceBox service is presented in Section 26.4, and Section 26.5 puts all the pieces together.

26.2 Introduction

The Service Configurator pattern [7] is a useful technique for configuring services and centralizing their administration. In practical terms, this means services are developed as dynamically-loadable components that can be configured into a general purpose “super server” in whatever combinations are necessary. IceBox is an implementation of the Service Configurator pattern for Ice services.

A generic IceBox server replaces the typical monolithic Ice server you normally write. The IceBox server is configured via properties with the application-specific services it is responsible for loading and managing, and it can be administered remotely. There are several advantages in using this architecture:

- Services loaded by the same IceBox server can be configured to take advantage of Ice's collocation optimizations. For example, if one service is a client of another service, and those services reside in the same IceBox server, then invocations between them can be optimized.
- Composing an application consisting of various services is done by configuration, not by compiling and linking. This decouples the service from the server, allowing services to be combined or separated as needed.
- Multiple Java services can be active in a single instance of a Java Virtual Machine (JVM). This conserves operating system resources when compared to running several monolithic servers, each in its own JVM.
- Services implement an IceBox service interface, providing a common framework for developers and a centralized administrative facility.
- IceBox support is integrated into IcePack, the server activation and deployment service (see Chapter 21).

26.3 The Service Manager

In addition to the objects supported by application services, an IceBox server supports an administrative object that implements the `IceBox::ServiceManager` interface (see Appendix B). This object is responsible for loading and initializing the services, as well as performing administrative actions requested by clients. An object adapter is created for this object so that its endpoint(s) can be secured against unauthorized access.

Currently, the administrative capabilities are rather limited: the only supported operation on the `ServiceManager` interface is `shutdown`, which terminates the services and shuts down the IceBox server. Additional administrative features may be added in a future release.

26.3.1 Endpoint Configuration

The endpoints for the service manager's object adapter are defined using the `IceBox.ServiceManager.Endpoints` configuration property:

```
IceBox.ServiceManager.Endpoints=tcp -p 10000
```

Since a malicious client could make a denial-of-service attack against the service manager, it is advisable to use SSL endpoints (see Chapter 24), or to secure these endpoints with the proper firewall configuration, or both.

26.3.2 Client Configuration

A client requiring administrative access to the service manager can create a proxy using the endpoints defined by the property described in Section 26.3.1. The default identity of the service manager object is `ServiceManager`, but this can be changed using the `IceBox.ServiceManager.Identity` property (see Appendix C). Therefore, a proxy using the default identity and the endpoint from Section 26.3.1 would be constructed as follows:

```
ServiceManager.Proxy=ServiceManager:tcp -p 10000
```

26.3.3 Administrative Utility

An administrative utility is included with IceBox. Given the limited capabilities of the `ServiceManager` interface in its present form, it should come as no surprise that the administrative utility also has limited functionality. IceBox provides C++ and Java implementations of the utility, with the same usage:

```
Usage: iceboxadmin [options] [command...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.

Commands:
shutdown            Shutdown the server.
```

The C++ utility is named `iceboxadmin`, while the Java utility is represented by the class `IceBox.Admin`. Both use the `IceBox.ServiceManager.Endpoints` property described in Section 26.3.1 to create the proxy for the service manager object. Typically, the utility is started using the same configuration file as the IceBox server (see Section 26.5), but that is not mandatory.

26.4 Developing a Service

Writing an IceBox service requires implementing one of the IceBox service interfaces. The C++ and Java sample implementations we present in this section implement `IceBox::Service`, shown below:

```
module IceBox {
local interface ServiceBase {
    void stop();
};
```

```
local interface Service extends ServiceBase {
    void start(string name,
               Ice::Communicator communicator,
               Ice::StringSeq args)
        throws FailureException;
};

};
```

As you can see, a service needs to implement only two operations, `start` and `stop`. These operations are invoked by the service manager; `start` is called after the service is loaded, and `stop` is called when the IceBox server is shutting down.

The `start` operation is the service's opportunity to initialize itself; this typically includes creating an object adapter and servants. The `name` and `args` parameters supply information from the service's configuration (see Section 26.4.3), and the `communicator` parameter is an `Ice::Communicator` object created by the service manager for use by the service. Depending on the service configuration, this `communicator` instance may be shared by other services in the same IceBox server, therefore care should be taken to ensure that items such as object adapters are given unique names.

The `stop` operation must reclaim any resources used by the service. Generally, a service deactivates its object adapter, and may also need to invoke `waitForDeactivate` on the object adapter in order to ensure that all pending requests have been completed before the clean up process can proceed. The service manager is responsible for destroying the `communicator`.

These interfaces are declared as `local` for a reason: they represent a contract between the service manager and the service, and are not intended to be used by remote clients. Any interaction the service has with remote clients is done via servants created by the service.

26.4.1 C++ Service Example

The example we present here is taken from the `demo/IceBox/Hello` sample program provided in the Ice distribution.

The class definition for our service is quite straightforward, but there are a few aspects worth mentioning:

```
#include <IceBox/IceBox.h>

#if defined(_WIN32)
#  define HELLO_API __declspec(dllexport)
```

```
#else
#  define HELLO_API /**
#endif

class HELLO_API HelloServiceI : public IceBox::Service {
public:
    virtual void start(const std::string &,
                       const Ice::CommunicatorPtr &,
                       const Ice::StringSeq &);
    virtual void stop();

private:
    Ice::ObjectAdapterPtr _adapter;
};
```

First, we include the `IceBox` header file so that we can derive our implementation from `IceBox::Service`.

Second, the preprocessor definitions are necessary because, on Windows, this service resides in a Dynamic Link Library (DLL), therefore we need to export the class so that the service manager can load it properly.

The member definitions are equally straightforward:

```
#include <Ice/Ice.h>
#include <HelloServiceI.h>
#include <HelloI.h>

using namespace std;

extern "C" {
    HELLO_API IceBox::Service *
    create(Ice::CommunicatorPtr communicator)
    {
        return new HelloServiceI;
    }
}

void
HelloServiceI::start(
    const string & name,
    const Ice::CommunicatorPtr & communicator,
    const Ice::StringSeq & args)
{
    _adapter = communicator->createObjectAdapter(name);
    Ice::ObjectPtr object = new HelloI(communicator);
    _adapter->add(object, Ice::stringToIdentity("hello"));
}
```

```
    _adapter->activate();
}

void
HelloServiceI::stop()
{
    _adapter->deactivate();
}
```

You might be wondering about the `create` function we defined. This is the *entry point* for a C++ IceBox service; that is, this function is used by the service manager to obtain an instance of the service, therefore it must have a particular signature. The name of the function is not important, but the function is expected to take a single argument of type `Ice::CommunicatorPtr`, and return a pointer to `IceBox::Service`¹. In this case, we simply return a new instance of `HelloServiceI`. See Section 26.4.3 for more information on entry points.

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

This is obviously a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the code into a shared library or DLL, it can be configured into an IceBox server as described in Section 26.4.3.

26.4.2 Java Service Example

As with the C++ example presented in the previous section, the complete source for the Java example can be found in the `demo/IceBox/hello` directory of the Ice distribution. The class definition for our service looks as follows:

```
public class HelloServiceI extends Ice.LocalObjectImpl
    implements IceBox.Service
{
    public void
    start(String name,
          Ice.Communicator communicator,
          String[] args)
        throws IceBox.FailureException
```

1. A function with C linkage cannot return an object type, such as a smart pointer, therefore the entry point must return a regular pointer value.

```
{  
    _adapter = communicator.createObjectAdapter(name);  
    Ice.Object object = new HelloI(communicator);  
    _adapter.add(object, Ice.Util.stringToIdentity("hello"));  
    _adapter.activate();  
}  
  
public void  
stop()  
{  
    _adapter.deactivate();  
}  
  
private Ice.ObjectAdapter _adapter;  
}
```

First, notice that our class extends `Ice.LocalObjectImpl`. This is one of the few occasions when an Ice developer must implement a local interface (other common cases are object factories and servant locators).

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

The service manager requires a service implementation to have a default constructor. This is the *entry point* for a Java IceBox service; that is, the service manager dynamically loads the service implementation class and invokes the default constructor to obtain an instance of the service.

This is obviously a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the service implementation class, it can be configured into an IceBox server as described in Section 26.4.3.

26.4.3 Configuring a Service

A service is configured into an IceBox server using a single property. This property serves several purposes: it defines the name of the service, it provides the service manager with the service entry point, and it defines properties and arguments for the service.

The format of the property is shown below:

```
IceBox.Service.name=entry_point [args]
```

The `name` component of the property key is the service name. This name is passed to the service's `start` operation, and must be unique among all services

configured in the same IceBox server. It is possible, though rarely necessary, to load two or more instances of the same service under different names.

The first argument in the property value is the entry point specification. For C++ services, this must have the form `library:symbol`, where `library` is the simple name of the service shared library or DLL, and `symbol` is the name of the entry point function. By simple name, we mean a name without any platform-specific prefixes or extensions; the service manager appends the appropriate decorations depending on the platform. For example, the simple name `MyService` results in a DLL named `MyService.dll` on Windows, and a shared library named `libMyService.so` on Linux. The shared library or DLL must reside in a directory that appears in `PATH` on Windows or `LD_LIBRARY_PATH` on POSIX systems.

For Java services, the entry point is simply the complete class name (including any package) of the service implementation class. The class must reside in the class path of the IceBox server.

Any arguments following the entry point specification are examined. If an argument has the form `--name=value`, then it is interpreted as a property definition that appears in the property set of the communicator passed to the service `start` operation. These arguments are removed, and any remaining arguments are passed to the `start` operation in the `args` parameter.

C++ Example

Here is an example of a configuration for our C++ example from Section 26.4.1:

```
IceBox.Service.Hello=HelloService:create \
--Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in `HelloService.dll` on Windows or `libHelloService.so` on Linux, and the entry point function `create` is invoked to create an instance of the service. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

Java Example

Here is an example of a configuration for our Java example from Section 26.4.2:

```
IceBox.Service.Hello=HelloServiceI \
--Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in the class `HelloService1`. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

Freeze Configuration

A Freeze service supports an additional configuration property that defines the pathname of the directory in which the service manager creates the Freeze database environment:

```
IceBox.DBEnvName.name=path
```

The `name` component of the property key is the service name.

Sharing a Communicator

The service manager can be configured to share a single communicator instance with all of its services using the following property:

```
IceBox.UseSharedCommunicator=1
```

The default behavior if this property is not specified is to create a new communicator instance for each service. However, if collocation optimizations between services are desired, then a shared communicator instance is required.

Loading Services

By default, the service manager loads the configured services in an undefined order, meaning services in the same IceBox server should not depend on one another. If services must be loaded in a particular order, the `IceBox.LoadOrder` property can be used:

```
IceBox.LoadOrder=Service1,Service2
```

In this example, `Service1` is loaded first, followed by `Service2`. Any remaining services are loaded after `Service2`, in an undefined order. Each service mentioned in `IceBox.LoadOrder` must have a matching `IceBox.Service` property.

26.4.4 Freeze Service

IceBox provides another interface for services that use Freeze (see Chapter 22):

```
module IceBox {  
    local interface FreezeService extends ServiceBase {  
        void start(string name,  
                  Ice::Communicator communicator,  
                  Ice::StringSeq args,  
                  Freeze::DBEnvironment dbEnv)  
            throws FailureException;  
    };  
};
```

By implementing `FreezeService` instead of `Service`, you have implicitly informed the service manager that it should create a `Freeze` database environment for this service and supply it to the `start` operation. The pathname of the database environment directory is specified using a configuration property, as described in Section 26.4.3. The service manager owns the database environment, and is responsible for destroying it after invoking `stop` on the service.

26.5 Starting IceBox

Incorporating everything we discussed in the previous sections, we can now configure and start IceBox servers.

26.5.1 Starting the C++ Server

The configuration file for our example C++ service is shown below:

```
IceBox.ServiceManager.Endpoints=tcp -p 10000  
IceBox.Service.Hello=HelloService:create  
Hello.Endpoints=tcp -p 10001
```

Notice that we define an endpoint for the object adapter created by the `Hello` service.

Assuming these properties reside in a configuration file named `config`, we can start the C++ IceBox server as follows:

```
$ icebox --Ice.Config=config
```

26.5.2 Starting the Java Server

Our Java configuration is nearly identical to the C++ version, except for the entry point specification:

```
IceBox.ServiceManager.Endpoints=tcp -p 10000
IceBox.Service.Hello=HelloServiceI
Hello.Endpoints=tcp -p 10001
```

Assuming these properties reside in a configuration file named `config`, we can start the Java IceBox server as follows:

```
$ java IceBox.Server --Ice.Config=config
```

26.5.3 Initialization Failure

At startup, an IceBox server inspects its configuration for all properties having the prefix `IceBox.Service.` and initializes each service. If initialization fails for a service, the IceBox server invokes the `stop` operation on any initialized services, reports an error, and terminates.

26.6 Summary

IceBox offers a refreshing change of perspective: developers focus on writing services, not applications. The definition of an application changes as well; using IceBox, an application becomes a collection of discrete services whose composition is determined dynamically by configuration, rather than statically by the linker.

Chapter 27

IceStorm

27.1 Chapter Overview

In this chapter we present IceStorm, an efficient publish/subscribe service for Ice applications. Section 27.2 provides an introduction to IceStorm, while Section 27.3 discusses some basic IceStorm concepts. An overview of the IceStorm Slice interfaces is provided in Section 27.4, and Section 27.5 presents an example IceStorm application. The IceStorm administration tool is described in Section 27.6, and the subject of federation is discussed in Section 27.7. IceStorm’s quality of service parameters are defined in Section 27.8. Finally, IceStorm configuration is addressed in Section 27.9.

27.2 Introduction

Applications often need to disseminate information to multiple recipients. For example, suppose we are developing a weather monitoring application in which we collect measurements such as wind speed and temperature from a meteorolog-

ical tower and periodically distribute them to weather monitoring stations. We initially consider using the architecture shown in Figure 27.1.

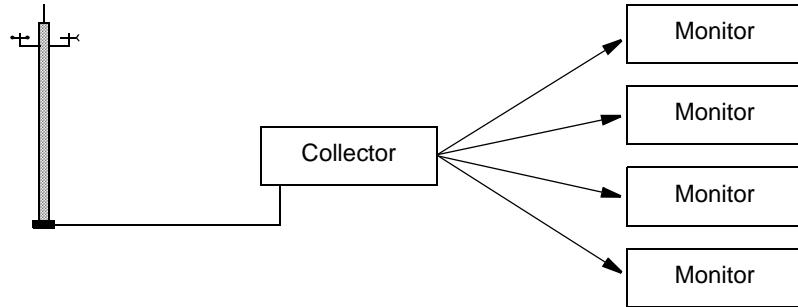


Figure 27.1. Initial design for a weather monitoring application.

However, the primary disadvantage of this architecture is that it tightly couples the collector to its monitors, needlessly complicating the collector implementation by requiring it to manage the details of monitor registration, measurement delivery, and error recovery. We can rid ourselves of these mundane duties by incorporating IceStorm into our architecture, as shown in Figure 27.2.

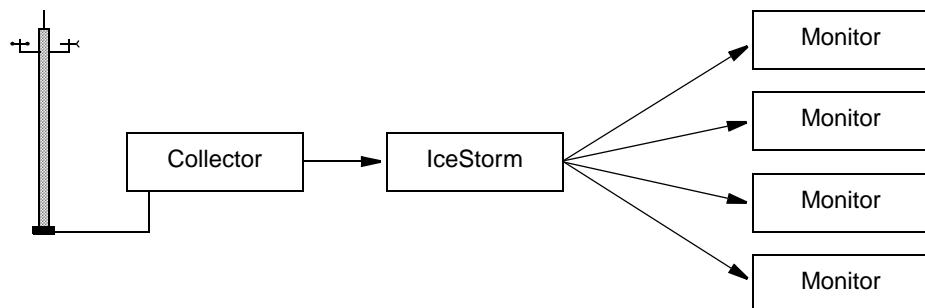


Figure 27.2. A weather monitoring application using IceStorm.

IceStorm simplifies the collector implementation significantly by decoupling it from the monitors. As a publish/subscribe service, IceStorm acts as a mediator between the collector (the publisher) and the monitors (the subscribers), and offers several advantages:

- When the collector is ready to distribute a new set of measurements, it makes a single request to the IceStorm server. The IceStorm server takes responsibility for delivering the request to the monitors, including handling any exceptions caused by ill-behaved or missing subscribers. The collector no longer needs to be aware of its monitors, or whether it even has any monitors at that moment.
- Similarly, monitors interact with the IceStorm server to perform tasks such as subscribing and unsubscribing, thereby allowing the collector to focus on its application-specific responsibilities and not on administrative trivia.
- The collector and monitor applications require very few changes to incorporate IceStorm.

27.3 Concepts

This section discusses several concepts that are important for understanding IceStorm’s capabilities.

27.3.1 Message

An IceStorm *message* is strongly typed and is represented by an invocation of a Slice operation: the operation name identifies the type of the message, and the operation parameters define the message contents. A message is published by invoking the operation on an IceStorm proxy in the normal fashion. Similarly, subscribers receive the message as a regular servant upcall. As a result, IceStorm uses the “push” model for message delivery; polling is not supported.

27.3.2 Topic

An application indicates its interest in receiving messages by subscribing to a *topic*. An IceStorm server supports any number of topics, which are created dynamically and distinguished by unique names. Each topic can have multiple publishers and subscribers.

A topic is essentially equivalent to an application-defined Slice interface: the operations of the interface define the types of messages supported by the topic. A publisher uses a proxy for the topic interface to send its messages, and a subscriber implements the topic interface (or an interface derived from the topic interface) in order to receive the messages. This is no different than if the

publisher and subscriber were communicating directly in the traditional client-server style; the interface represents the contract between the client (the publisher) and the server (the subscriber), except IceStorm transparently forwards each message to multiple recipients.

IceStorm does not verify that publishers and subscribers are using compatible interfaces, therefore applications must ensure that topics are used correctly.

27.3.3 Oneway

IceStorm messages have oneway semantics (see Section 2.2.2), therefore a publisher cannot receive replies from its subscribers. Naturally, all of the Ice transports (TCP, SSL, and UDP) can be used to publish and receive messages.

27.3.4 Federation

IceStorm supports the formation of topic graphs, also known as federation. A topic graph is formed by creating links between topics, where a *link* is a unidirectional association from one topic to another. Each link has a *cost* that may restrict message delivery on that link (see Section 27.7.2). A message published on a topic is also published on all of the topic's links for which the message cost does not exceed the link cost.

Once a message has been published on a link, the receiving topic publishes the message to its subscribers, but does not publish it on any of its links. In other words, IceStorm messages propagate at most one hop from the originating topic in a federation (see Section 27.7.1).

Figure 27.3 presents an example of topic federation. Topic T_1 has links to T_2 and T_3 , as indicated by the arrows. The subscribers S_1 and S_2 receive all messages

published on T_2 , as well as those published on T_1 . Subscriber S_3 receives messages only from T_1 , and S_4 receives messages from both T_3 and T_1 .

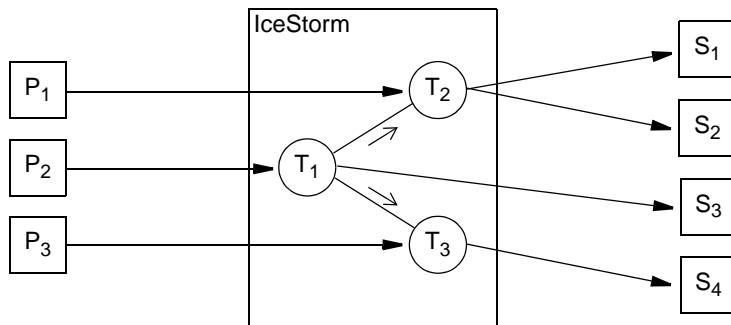


Figure 27.3. Topic federation.

IceStorm makes no attempt to prevent a subscriber from receiving duplicate messages. For example, if a subscriber is subscribed to both T_2 and T_3 , then it would receive two requests for each message published on T_1 .

27.3.5 Quality of Service

IceStorm allows each subscriber to specify its own *quality of service* (QoS) parameters that affect the delivery of its messages. Quality of service parameters are represented as a dictionary of name-value pairs. The supported QoS parameters are described in Section 27.8.

27.3.6 Persistence

IceStorm has a database in which it maintains information about its topics and links. However, a message sent via IceStorm is not stored persistently, but rather is discarded as soon as it is delivered to the topic's current set of subscribers. If an error occurs during delivery to a subscriber, IceStorm does not queue messages for that subscriber.

27.3.7 Subscriber Errors

Since IceStorm messages are delivered with oneway semantics, the only errors IceStorm can detect are connection or timeout failures. If IceStorm encounters

such a failure while attempting to deliver a message to a subscriber, the subscriber is immediately unsubscribed from the topic on which the message was published.

Slow subscribers can also create problems. IceStorm attempts to deliver messages as soon as they are published: when the IceStorm publisher object receives a message, it may immediately make nested invocations to the topic's subscribers. Therefore, if a subscriber is not consuming messages as fast as they are being published, it may cause threads to accumulate in IceStorm. If the number of threads reaches the maximum size of the thread pool, then no new messages can be published. See Chapter 15 for more information on Ice thread pools.

27.4 IceStorm Interface Overview

This section provides a brief introduction to the Slice interfaces comprising the IceStorm service. See Appendix B for the Slice documentation.

27.4.1 TopicManager

The TopicManager is a singleton object that acts as a factory and repository of Topic objects. Its interface and related types are shown below:

```
module IceStorm {
    dictionary<string, Topic*> TopicDict;

    exception TopicExists {
        string name;
    };

    exception NoSuchTopic {
        string name;
    };

    interface TopicManager {
        Topic* create(string name) throws TopicExists;
        nonmutating Topic* retrieve(string name) throws NoSuchTopic;
        nonmutating TopicDict retrieveAll();
    };
}
```

The `create` operation is used to create a new topic, which must have a unique name. The `retrieve` operation allows a client to obtain a proxy for an existing topic, and `retrieveAll` supplies a dictionary of all existing topics.

27.4.2 Topic

The `Topic` interface represents a topic and provides several administrative operations for configuring links and managing subscribers.

```
module IceStorm {
    struct LinkInfo {
        Topic* theTopic;
        string name;
        int cost;
    };
    sequence<LinkInfo> LinkInfoSeq;

    dictionary<string, string> QoS;

    exception LinkExists {
        string name;
    };

    exception NoSuchLink {
        string name;
    };

    interface Topic {
        nonmutating string getName();
        nonmutating Object* getPublisher();
        void subscribe(QoS theQoS, Object* subscriber);
        idempotent void unsubscribe(Object* subscriber);
        idempotent void link(Topic* linkTo, int cost)
            throws LinkExists;
        idempotent void unlink(Topic* linkTo) throws NoSuchLink;
        nonmutating LinkInfoSeq getLinkInfoSeq();
        void destroy();
    };
}
```

The `getName` operation returns the name assigned to the topic, while the `getPublisher` operation returns a proxy for the topic's publisher object (see Section 27.5.2).

The `subscribe` operation adds a subscriber's proxy to the topic; if another subscriber proxy already exists with the same object identity, the subscriber's proxy is replaced with the new one. The `unsubscribe` operation removes a subscriber from the topic.

A link to another topic is created using the `link` operation; if a link already exists to the given topic, the `LinkExists` exception is raised. Links are destroyed using the `unlink` operation.

Finally, the `destroy` operation permanently destroys the topic.

27.5 Using IceStorm

In this section we expand on the weather monitoring example from Section 27.2, demonstrating how to create, subscribe to and publish messages on a topic. We use the following Slice definitions in our example:

```
struct Measurement {
    string tower; // tower id
    float windSpeed; // knots
    short windDirection; // degrees
    float temperature; // degrees Celsius
};

interface Monitor {
    void report(Measurement m);
};
```

`Monitor` is our topic interface. For the sake of simplicity, it defines just one operation, `report`, taking a `Measurement` struct as its only parameter.

27.5.1 Implementing a Publisher

The implementation of our collector application can be summarized easily:

1. Obtain a proxy for the `TopicManager`. This is the primary IceStorm object, used by both publishers and subscribers.
2. Obtain a proxy for the `Weather` topic, either by creating the topic if it does not exist, or retrieving the proxy for the existing topic.
3. Obtain a proxy for the `Weather` topic's “publisher object.” This proxy is provided for the purpose of publishing messages, and therefore is narrowed to the topic interface (`Monitor`).

4. Collect and report measurements.

In the sections below, we present collector implementations in C++ and Java.

C++ Example

As usual, our C++ example begins by including the necessary header files. The interesting ones are `IceStorm/IceStorm.h`, which is generated from the `IceStorm` Slice definitions, and `Monitor.h`, containing the generated code for our monitor definitions shown above.

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

int main(int argc, char* argv[])
{
    ...
    Ice::ObjectPrx obj = communicator->stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm::TopicManagerPrx topicManager =
        IceStorm::TopicManagerPrx::checkedCast(obj);
    IceStorm::TopicPrx topic;
    try {
        topic = topicManager->retrieve("Weather");
    }
    catch (const IceStorm::NoSuchTopic&)
    {
        topic = topicManager->create("Weather");
    }

    Ice::ObjectPrx pub = topic->getPublisher();
    if (!pub->ice_isDatagram())
        pub = pub->ice_oneway();
    MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
    while (true) {
        Measurement m = getMeasurement();
        monitor->report(m);
    }
    ...
}
```

After obtaining a proxy for the topic manager, the collector attempts to retrieve the topic. If the topic does not exist yet, the collector receives a `NoSuchTopic` exception and then creates the topic.

```

IceStorm::TopicPrx topic;
try {
    topic = topicManager->retrieve("Weather");
}
catch (const IceStorm::NoSuchTopic&) {
    topic = topicManager->create("Weather");
}

```

The next step is obtaining a proxy for the publisher object, which the collector narrows to the `Monitor` interface.

```

Ice::ObjectPrx pub = topic->getPublisher();
if (!pub->ice_isDatagram())
    pub = pub->ice_oneway();
MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);

```

Finally, the collector enters its main loop, collecting measurements and publishing them via the IceStorm publisher object.

```

while (true) {
    Measurement m = getMeasurement();
    monitor->report(m);
}

```

Java Example

The equivalent Java version is shown below.

```

public static void main(String[] args)
{
    ...
    Ice.ObjectPrx obj = communicator.stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm.TopicManagerPrx topicManager =
        IceStorm.TopicManagerPrxHelper.checkedCast(obj);
    IceStorm.TopicPrx topic = null;
    try {
        topic = topicManager.retrieve("Weather");
    }
    catch (IceStorm.NoSuchTopic ex) {
        topic = topicManager.create("Weather");
    }

    Ice.ObjectPrx pub = topic.getPublisher();
    if (!pub.ice_isDatagram())
        pub = pub.ice_oneway();
    MonitorPrx monitor = MonitorPrxHelper.uncheckedCast(pub);
    while (true) {

```

```
        Measurement m = getMeasurement();
        monitor.report(m);
    }
    ...
}
```

After obtaining a proxy for the topic manager, the collector attempts to retrieve the topic. If the topic does not exist yet, the collector receives a `NoSuchTopic` exception and then creates the topic.

```
IceStorm.TopicPrx topic = null;
try {
    topic = topicManager.retrieve("Weather");
}
catch (IceStorm.NoSuchTopic ex) {
    topic = topicManager.create("Weather");
}
```

The next step is obtaining a proxy for the publisher object, which the collector narrows to the `Monitor` interface.

```
Ice.ObjectPrx pub = topic.getPublisher();
if (!pub.ice_isDatagram())
    pub = pub.ice_oneway();
MonitorPrx monitor = MonitorPrxHelper.uncheckedCast(pub);
```

Finally, the collector enters its main loop, collecting measurements and publishing them via the IceStorm publisher object.

```
while (true) {
    Measurement m = getMeasurement();
    monitor.report(m);
}
```

27.5.2 Using a Publisher Object

Each topic creates a publisher object for the express purpose of publishing messages. It is a special object in that it implements an Ice interface that allows the object to receive and forward requests (i.e., IceStorm messages) without requiring knowledge of the operation types.

From the publisher's perspective, the publisher object appears to be an application-specific type. In reality, the publisher object can forward requests for any type, and that introduces a degree of risk: a misbehaving publisher can use `uncheckedCast` to narrow the publisher object to any type and invoke any operation; the publisher object unknowingly forwards those requests to the

subscribers. If the publisher sends a request using an incorrect type, the Ice run time in a subscriber typically responds by raising `OperationNotExistException`. However, since the subscriber receives its messages as oneway invocations, no response can be sent to the publisher object to indicate this failure, and therefore neither the publisher nor the subscriber is aware of the type-mismatch problem. In short, IceStorm places the burden on the developer to ensure that publishers and subscribers are using it correctly.

Although it is not strictly necessary for a publisher to use a oneway proxy for the publisher object, there are two reasons why it is recommended:

- it reiterates the fact that no replies are possible using this proxy;
- it is more efficient because the Ice run time in the collector does not waste time waiting for a reply to each request.

The second reason is especially important, given how IceStorm delivers its messages. As mentioned in Section 27.3.7, the publisher object attempts to deliver a message immediately upon publication of the message, but does not guarantee that the message is delivered to all subscribers before the publication invocation returns. Therefore, a publisher gains nothing by making a twoway invocation on the publisher object, and risks unnecessarily delaying its own processing due to IceStorm implementation details.

Note that using a oneway proxy does not necessarily imply a loss of reliability. If the publisher endpoints use reliable transports such as TCP or SSL, oneway messages are always delivered reliably to the IceStorm server. However, each subscriber can select its own transport for message delivery, therefore the transport used between the publisher and the IceStorm server has no effect on subscribers.¹

27.5.3 Implementing a Subscriber

Our subscriber implementation takes the following steps:

1. Obtain a proxy for the `TopicManager`. This is the primary IceStorm object, used by both publishers and subscribers.
2. Create an object adapter to host our `Monitor` servant.

1. The UDP transport might be used for a subscriber's endpoint if the possibility of lost messages is acceptable. However, UDP is not typically used for a `TopicManager` endpoint because the publisher generally wants to ensure that messages are delivered reliably to IceStorm, even if they may not be delivered reliably to subscribers.

3. Instantiate the `Monitor` servant and activate it with the object adapter.
4. Subscribe to the `Weather` topic.
5. Process `report` messages until shutdown.
6. Unsubscribe from the `Weather` topic.

In the sections below, we present monitor implementations in C++ and Java.

C++ Example

Our C++ monitor implementation begins by including the necessary header files. The interesting ones are `IceStorm/IceStorm.h`, which is generated from the IceStorm Slice definitions, and `Monitor.h`, containing the generated code for our monitor definitions shown at the beginning of Section 27.2.

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

using namespace std;

class MonitorI : virtual public Monitor {
public:
    virtual void report(const Measurements& m,
                        const Ice::Current&) {
        cout << "Measurement report:" << endl
            << " Tower: " << m.tower << endl
            << " W Spd: " << m.windSpeed << endl
            << " W Dir: " << m.windDirection << endl
            << " Temp: " << m.temperature << endl
            << endl;
    }
};

int main(int argc, char* argv[])
{
    ...
    Ice::ObjectPrx obj = communicator->stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm::TopicManagerPrx topicManager =
        IceStorm::TopicManagerPrx::checkedCast(obj);

    Ice::ObjectAdapterPtr adapter =
        communicator->createObjectAdapter("MonitorAdapter");

    MonitorPtr monitor = new MonitorI;
```

```

Ice::ObjectPrx proxy = adapter->addWithUUID(monitor);

IceStorm::TopicPrx topic;
try {
    topic = topicManager->retrieve("Weather");
    IceStorm::QoS qos;
    topic->subscribe(qos, proxy);
}
catch (const IceStorm::NoSuchTopic&) {
    // Error! No topic found!
    ...
}

adapter->activate();
communicator->waitForShutdown();

topic->unsubscribe(proxy);
...
}

```

Our implementation of the `Monitor` servant is currently quite simple. A real implementation might update a graphical display, or incorporate the measurements into an ongoing calculation.

```

class MonitorI : virtual public Monitor {
public:
    virtual void report(const Measurement& m,
                        const Ice::Current&) {
        cout << "Measurement report:" << endl
        << " Tower: " << m.tower << endl
        << " W Spd: " << m.windSpeed << endl
        << " W Dir: " << m.windDirection << endl
        << " Temp: " << m.temperature << endl
        << endl;
    }
};

```

After obtaining a proxy for the topic manager, the program creates an object adapter, instantiates the `Monitor` servant and activates it.

```

Ice::ObjectAdapterPtr adapter =
    communicator->createObjectAdapter("MonitorAdapter");

MonitorPtr monitor = new MonitorI;
Ice::ObjectPrx proxy = adapter->addWithUUID(monitor);

```

Next, the monitor subscribes to the topic.

```
IceStorm::TopicPrx topic;
try {
    topic = topicManager->retrieve("Weather");
    IceStorm::QoS qos;
    topic->subscribe(qos, proxy);
}
catch (const IceStorm::NoSuchTopic&) {
    // Error! No topic found!
    ...
}
```

Finally, the monitor activates its object adapter and waits to be shutdown. After `waitForShutdown` returns, the monitor cleans up by unsubscribing from the topic.

```
adapter->activate();
communicator->waitForShutdown();

topic->unsubscribe(proxy);
```

Java Example

The Java implementation of the monitor is shown below.

```
class MonitorI extends _MonitorDisp {
    public void report(Measurement m, Ice.Current curr) {
        System.out.println(
            "Measurement report:\n" +
            "  Tower: " + m.tower + "\n" +
            "  W Spd: " + m.windSpeed + "\n" +
            "  W Dir: " + m.windDirection + "\n" +
            "  Temp: " + m.temperature + "\n");
    }
}

public static void main(String[] args)
{
    ...
    Ice.ObjectPrx obj = communicator.stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm.TopicManagerPrx topicManager =
        IceStorm.TopicManagerPrxHelper.checkedCast(obj);

    Ice.ObjectAdapterPtr adapter =
        communicator.createObjectAdapter("MonitorAdapter");

    Monitor monitor = new MonitorI();
```

```

Ice.ObjectPrx proxy = adapter.addWithUUID(monitor);

IceStorm.TopicPrx topic = null;
try {
    topic = topicManager.retrieve("Weather");
    java.util.Map qos = null;
    topic.subscribe(qos, proxy);
}
catch (IceStorm.NoSuchTopic ex) {
    // Error! No topic found!
    ...
}

adapter.activate();
communicator.waitForShutdown();

topic.unsubscribe(proxy);
...
}

```

Our implementation of the `Monitor` servant is currently quite simple. A real implementation might update a graphical display, or incorporate the measurements into an ongoing calculation.

```

class MonitorI extends _MonitorDisp {
    public void report(Measurement m, Ice.Current curr) {
        System.out.println(
            "Measurement report:\n" +
            "  Tower: " + m.tower + "\n" +
            "  W Spd: " + m.windSpeed + "\n" +
            "  W Dir: " + m.windDirection + "\n" +
            "  Temp: " + m.temperature + "\n");
    }
}

```

After obtaining a proxy for the topic manager, the program creates an object adapter, instantiates the `Monitor` servant and activates it.

```

Monitor monitor = new MonitorI();
Ice.ObjectPrx proxy = adapter.addWithUUID(monitor);

```

Next, the monitor subscribes to the topic.

```

IceStorm.TopicPrx topic = null;
try {
    topic = topicManager.retrieve("Weather");
    java.util.Map qos = null;

```

```
        topic.subscribe(qos, proxy);
    }
    catch (IceStorm.NoSuchTopic ex) {
        // Error! No topic found!
        ...
    }
```

Finally, the monitor activates its object adapter and waits to be shutdown. After `waitForShutdown` returns, the monitor cleans up by unsubscribing from the topic.

```
adapter.activate();
communicator.waitForShutdown();

topic.unsubscribe(proxy);
```

27.6 IceStorm Administration

The IceStorm administration tool is a command-line program that provides administrative control of an IceStorm server. The tool requires that the `IceStorm.TopicManager.Proxy` property be specified as described in Section 27.9.2.

The following command-line options are supported:

```
$ icestormadmin -h
Usage: icestormadmin [options] [file...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
-DNAME              Define NAME as 1.
-DNAME=DEF          Define NAME as DEF.
-UNAME              Remove any definition for NAME.
-IDIR               Put DIR in the include file search path.
-e COMMANDS         Execute COMMANDS.
-d, --debug          Print debug messages.
```

The tool operates in three modes, depending on the command-line arguments:

1. If one or more `-e` options are specified, the tool executes the given commands and exits.
2. If one or more files are specified, the tool preprocesses each file with the C preprocessor, executes the commands in each file, and exits.
3. Otherwise, the tool enters an interactive session.

The **help** command displays the following usage information:

help

Print this message.

exit, quit

Exit this program.

create TOPICS

Add TOPICS.

destroy TOPICS

Remove TOPICS.

link FROM TO COST

Link FROM to TO with the given COST.

unlink FROM TO

Unlink TO from FROM.

graph DATA COST

Construct the link graph as described in DATA with COST.

list [TOPICS]

Display information on TOPICS or all topics.

Many of the commands accept one or more topic names (**TOPICS**) as arguments. Topic names containing white space or matching a command keyword must be enclosed in single or double quotes.

For more information on the **graph** command, see Section 27.7.3.

27.7 Topic Federation

The ability to link topics together into a federation provides IceStorm applications with a lot of flexibility, while the notion of a “cost” associated with links allows applications to restrict the flow of messages in creative ways. IceStorm applications have complete control of topic federation using the `TopicManager` interface described in Appendix B, allowing links to be created and removed dynamically as necessary. For many applications, however, the topic graph is static and therefore can be configured using the administrative tool discussed in Section 27.6.

27.7.1 Message Propagation

IceStorm messages are never propagated over more than one link. For example, consider the topic graph shown in Figure 27.4.

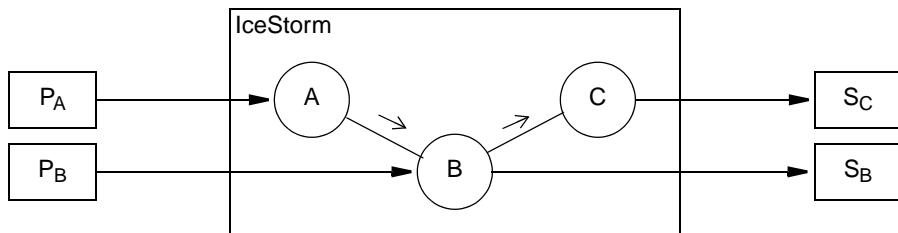


Figure 27.4. Message propagation.

In this case, messages published on A are propagated to B, but B does not propagate A's messages to C. Therefore, subscriber S_B receives messages published on topics A and B, but subscriber S_C only receives messages published on topics B and C. If the application needs messages to propagate from A to C, then a link must be established directly between A and C.

27.7.2 Cost

As described in Section 27.7.1, IceStorm messages are only propagated on the originating topic's immediate links. In addition, applications can use the notion of cost to further restrict message propagation.

A cost is associated with messages and links. When a message is published on a topic, the topic compares the cost associated with each of its links against the message cost, and only propagates the message on those links whose cost equals or exceeds the message cost. A cost value of zero (0) has the following implications:

- messages with a cost value of zero (0) are published on all of the topic's links regardless of the link cost;
- links with a cost value of zero (0) accept all messages regardless of the message cost.

For example, consider the topic graph shown in Figure 27.5.

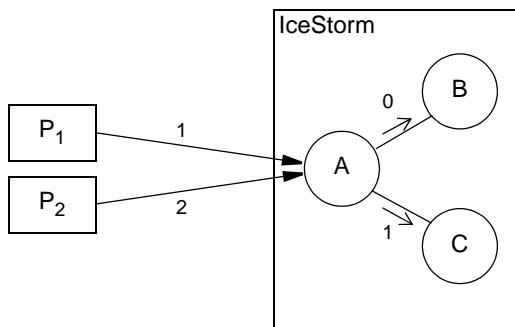


Figure 27.5. Cost semantics.

Publisher P_1 publishes a message on topic A with a cost of 1. This message is propagated on the link to topic B because the link has a cost of 0 and therefore accepts all messages. The message is also propagated on the link to topic C , because the message cost does not exceed the link cost (1). On the other hand, the message published by P_2 with a cost of 2 is only propagated on the link to B .

Request Context

The cost of a message is specified in an Ice request context. Each Ice proxy operation has an implicit argument of type `Ice::Context` representing the request context (see Section 16.8). This argument is rarely used, but it is the ideal location for specifying the cost of an IceStorm message because an application only needs to supply a request context if it actually uses IceStorm's cost feature. If the request context does not contain a cost value, the message is assigned the default cost value of zero (0).

Publishing a Message with a Cost

The code examples below demonstrate how a collector can publish a measurement with a cost value of 5. First, the C++ version:

```

Measurement m = getMeasurement();
Ice::Context ctx;
ctx["cost"] = "5";
monitor->report(m, ctx);

```

And here is the equivalent version in Java:

```
Measurement m = getMeasurement();
java.util.HashMap ctx = new java.util.HashMap();
ctx.put("cost", "5");
monitor.report(m, ctx);
```

Receiving a Message with a Cost

A subscriber can discover the cost of a message by examining the request context supplied in the `Ice::Current` argument. For example, here is a C++ implementation of `Monitor::report` that displays the cost value if it is present:

```
virtual void report(const Measurement& m,
                     const Ice::Current& curr) {
    Ice::Context::const_iterator p = curr.ctx.find("cost");
    cout << "Measurement report:" << endl
        << " Tower: " << m.tower << endl
        << " W Spd: " << m.windSpeed << endl
        << " W Dir: " << m.windDirection << endl
        << " Temp: " << m.temperature << endl
        << " Temp: " << m.temperature << endl;
    if (p != curr.ctx.end())
        cout << " Cost: " << p->second << endl;
    cout << endl;
}
```

And here is the equivalent Java implementation:

```
public void report(Measurement m, Ice.Current curr) {
    String cost = null;
    if (curr.ctx != null)
        cost = curr.ctx.get("cost");
    System.out.println(
        "Measurement report:\n" +
        " Tower: " + m.tower + "\n" +
        " W Spd: " + m.windSpeed + "\n" +
        " W Dir: " + m.windDirection + "\n" +
        " Temp: " + m.temperature);
    if (cost != null)
        System.out.println(" Cost: " + cost);
    System.out.println();
}
```

For the sake of efficiency, the Ice for Java run time may supply a null value for the request context in `Ice.Current`, therefore an application is required to check for null before using the request context. Furthermore, a non-null request context

is subject to change after the completion of the request, so an application that wishes to retain the request context must make a copy of it (e.g., using `clone`).

27.7.3 Automating Federation

Given the restrictions on message propagation described in the previous sections, creating a complex topic graph can be a tedious endeavor. Of course, creating a topic graph is not typically a common occurrence, since IceStorm keeps a persistent record of the graph. However, there are situations where an automated procedure for creating a topic graph can be valuable, such as during development when the graph might change significantly and often, or when graphs need to be recomputed based on changing costs.

Administration Tool Script

A simple way to automate the creation of a topic graph is to create a text file containing commands to be executed by the IceStorm administration tool. For example, the commands to create the topic graph shown in Figure 27.5 are shown below:

```
create A B C
link A B 0
link A C 1
```

If we store these commands in the file `graph.txt`, we can execute them using the following command:

```
$ icestormadmin --Ice.Config=config graph.txt
```

We assume that the configuration file `config` contains the definition for the property `IceStorm.TopicManager.Proxy`.

XML Graph Descriptor

IceStorm provides an alternative method of configuring topic graphs for applications that use message and link costs. This feature is employed using the administration tool's `graph` command, which accepts an XML graph descriptor and

establishes links between topics based on reachability within a maximum cost. For example, let us start with the graph shown in Figure 27.6.

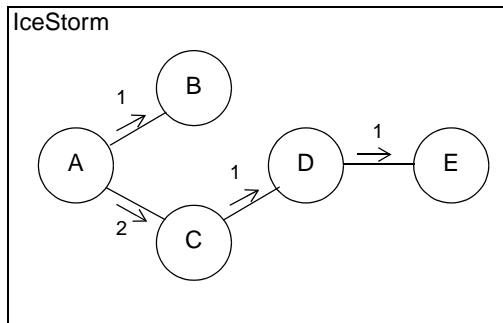


Figure 27.6. Initial graph.

The XML graph descriptor for this topic graph is shown below.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<graph>
    <vertex-set>
        <vertex name="A" />
        <vertex name="B" />
        <vertex name="C" />
        <vertex name="D" />
        <vertex name="E" />
    </vertex-set>
    <edge-set>
        <edge source="A" target="B" cost="1"/>
        <edge source="A" target="C" cost="2"/>
        <edge source="C" target="D" cost="1"/>
        <edge source="D" target="E" cost="1"/>
    </edge-set>
</graph>
```

As you can see, the descriptor format is quite straightforward. The outer `graph` element contains one `vertex-set` element and one `edge-set` element. The `vertex-set` element is comprised of two or more `vertex` elements having a single attribute, `name`, providing the name of a topic. The `edge-set` element contains one or more `edge` elements that describe links between topics. The `source` and `target` attributes supply the topic names, and the `cost` attribute defines the link cost.

Although this descriptor accurately describes the structure of the graph in Figure 27.6, the graph that is ultimately created by the administration tool's **graph** command is dependent on the specified maximum cost. Suppose we save our graph descriptor as `graph.xml` and execute the following commands:

```
$ icestormadmin --Ice.Config=config  
>>> create A B C  
>>> graph "graph.xml" 1
```

We must first create the topics that are used as vertices in our XML graph descriptor. Next, we instruct the administration tool to create a graph using a maximum cost of 1, which results in the graph shown in Figure 27.7.

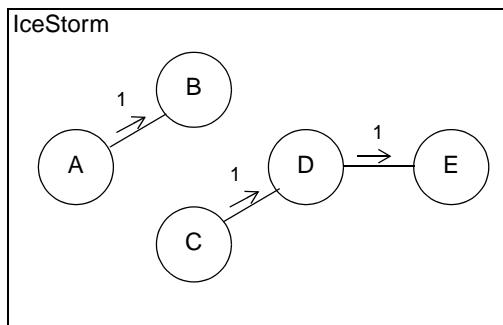


Figure 27.7. Graph with maximum cost of 1.

Notice that the link from A to C was not created, because the cost of that link (2) exceeded the maximum cost. Now consider the outcome of using a maximum cost of 2:

```
>>> graph "graph.xml" 2
```

As shown in Figure 27.8, the link from A to C is now present. In addition, a link from C to E has been established with a cost of 2, because E is reachable from C within the maximum cost.

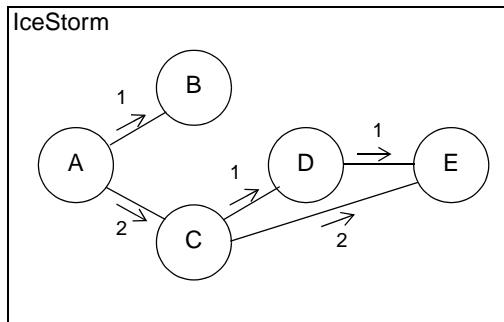


Figure 27.8. Graph with maximum cost of 2.

We can verify that the links are created properly using the **list** command:

```
>>> list A B C D E
A
    B with cost 0
    C with cost 2
B
C
    D with cost 1
    E with cost 2
D
    E with cost 1
E
```

The **graph** command can be used repeatedly to experiment with different cost configurations. For each invocation of the **graph** command, the administration tool recomputes the graph based on the specified maximum cost, establishes the necessary links, and removes any links that are no longer valid given the new cost constraints.

27.8 Quality of Service

An IceStorm subscriber specifies QoS parameters at the time of subscription. IceStorm currently supports only one QoS parameter, `reliability`, which is described in the section below.

27.8.1 Reliability

The QoS parameter `reliability` affects message delivery. The default value is `oneway`, in which each published message results in a separate `oneway` invocation to the subscriber. Alternatively, the value `batch` causes IceStorm to temporarily queue messages for the subscriber and flush them at regular intervals by delivering them as `batch oneway` requests (see Chapter 19). There is no difference between the two reliability modes as far as the subscriber's servant implementation is concerned; the messages are delivered individually as servant up-calls in the normal fashion. However, there is a difference in terms of safety and performance.

The `oneway` mode emphasizes safety by delivering its messages as soon as they are received in order to minimize the possibility of lost messages should an IceStorm failure occur.

By contrast, the `batch` mode queues messages for a configurable amount of time, increasing the possibility of lost messages in the event of failure, but providing better performance by minimizing network overhead. The frequency with which the `batch` mode flushes its queues is determined by the configuration property `IceStorm.Flush.Timeout`, as described in Appendix C.

The performance improvement offered by the `batch` mode is especially apparent for topics in which small messages are sent frequently. Rather than making many small requests, as would be done in `oneway` mode, the messages are accumulated and delivered in groups, amortizing the network overhead across several messages. However, the subscriber should only use the `batch` mode if it is willing to wait up to the flush timeout for new messages.

27.8.2 Example

The Slice type `IceStorm::QoS` is defined as a `dictionary` whose key and value types are both `string`, therefore the QoS parameter name and value are both represented as strings. The example code presented in Section 27.5.3 used an empty dictionary for the QoS argument, meaning default values are used. The

C++ and Java examples shown below illustrate how to set the `reliability` parameter to `batch`.

C++ Example

```
IceStorm::QoS qos;
qos["reliability"] = "batch";
topic->subscribe(qos, proxy);
```

Java Example

```
java.util.Map qos = new java.util.HashMap();
qos.put("reliability", "batch");
topic.subscribe(qos, proxy);
```

27.9 Configuring IceStorm

IceStorm is a relatively lightweight service in that it requires very little configuration and is implemented as an IceBox service (see Chapter 26). The configuration properties supported by IceStorm are described in Appendix C; however, most of them control diagnostic output and are not discussed in this chapter.

27.9.1 Server Configuration

The sample server configuration file shown below presents the properties of primary interest:

```
IceBox.Service.IceStorm=IceStormService,1.0.0:create
IceBox.DBEnvName.IceStorm=db
IceStorm.TopicManager.Endpoints=tcp -p 9999
IceStorm.Publish.Endpoints=tcp
```

The first property defines the entry point for the IceStorm service. The service name (the last component of the `IceBox.Service` property name, `IceStorm` in this example) determines the prefix for the IceStorm configuration properties. It is not mandatory to use `IceStorm` as the service name, but it is recommended if there is no preferred name.

IceStorm uses Freeze to manage the service's persistent state, therefore the second property specifies the pathname of the Freeze database environment directory (see Chapter 22) for the service. In this example, the directory `db` is used, which must already exist in the current working directory.

The final two properties specify the endpoints used by the IceStorm object adapters; notice that their property names begin with `IceStorm`, matching the service name. The `TopicManager` property specifies the endpoints on which the `TopicManager` and `Topic` objects reside; these endpoints typically use a connection-oriented protocol such as TCP or SSL. The `Publish` property specifies the endpoints used by topic publisher objects.

27.9.2 Client Configuration

Clients of the service can define a proxy for the `TopicManager` object as follows:

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 9999
```

The name of the property is not relevant, but the endpoint must match that of the `IceStorm.TopicManager.Endpoints` property, and the object identity must use the IceStorm service name as the identity category and `TopicManager` as the identity name.

27.10 Summary

IceStorm is a publish/subscribe service that offers Ice applications a flexible and efficient means of publishing oneway requests to a group of subscribers. IceStorm simplifies development by relieving the application from the burden of managing subscribers and handling delivery errors, allowing the application to focus on publishing its data and not on the minutiae of distribution. Finally, IceStorm leverages Ice request-forwarding facilities in order to provide a typed interface to publishers and subscribers, minimizing the impact on applications.

Chapter 28

IcePatch

28.1 Chapter Overview

This chapter presents IcePatch, the Ice solution for secure replication of a directory tree. Section 28.2 provides an overview of IcePatch concepts and operation. Section 28.3 discusses server configuration and usage, while Section 28.4 describes the IcePatch client. Section 28.5 presents the IcePatch client application programming interface, and Section 28.6 explains the use of download totals.

28.2 Introduction

IcePatch is an efficient file patching service that is easy to configure and use. It includes the following components:

- the IcePatch server
- a text-based IcePatch client
- a Slice API and C++ utility library for developing custom IcePatch clients

As with all Ice services, IcePatch can be configured to use Ice facilities such as Glacier for firewall support and IceSSL for secure communication.

IcePatch is conceptually quite simple. The server is given responsibility for a file system directory (the “data directory”) containing the files and subdirectories

that are to be distributed to IcePatch clients. The server computes a unique signature for each file and subdirectory, and creates a compressed version of each file in order to minimize bandwidth consumption.

An IcePatch client may patch with the top-level data directory, or only selected subdirectories. The first time the client requests a directory, the entire directory is transferred, including any subdirectories, and unique signatures are computed for the local files. Upon subsequent requests of the same directory, the client compares its local data directory against the server's and performs the following activities:

- transfers new files and directories that are not present in the client's local data directory,
- transfers files whose unique signatures do not match the server's, and
- optionally removes local files and directories that are no longer present in the server's data directory.

A signature mismatch between two files of the same name indicates that the file was updated in the server's data directory, in which case the entire file is downloaded to replace the local file. IcePatch does not attempt to compare the contents of the local and remote files in order to transfer only the differences between them. However, if a file transfer was terminated prematurely, IcePatch preserves the local file fragment and resumes the transfer from that point.

The IcePatch client always transfers a file in its compressed form. After the transfer completes, the client uncompresses the file, removes the compressed version, and computes a unique signature for the uncompressed file.

Whereas the unique signature of a file is computed using the file's contents, the unique signature of a directory is computed using the name and signature of each file and subdirectory it contains. This strategy allows the detection of any change in a directory, including the addition, modification, or removal of a file or subdirectory, as well as the renaming of a file or subdirectory. During patching, the client can safely skip an entire subdirectory if the unique signature of the local subdirectory matches that of the server.

28.3 The IcePatch Server

This section describes the configuration and use of the IcePatch server.

28.3.1 Configuring the Server

The IcePatch server supports a number of configuration properties, as described in Appendix C. A minimal configuration is shown below:

```
IcePatch.Endpoints=tcp -p 10000
IcePatch.Directory=/home/icepatch/data
IcePatch.UpdatePeriod=60
IcePatch.BusyTimeout=15
IcePatch.Trace.Files=0
```

The `IcePatch.Endpoints` property specifies one or more endpoints on which the server listens for requests from clients. This is the only mandatory property.

The `IcePatch.Directory` property specifies the server's data directory. If not defined, the server uses its current working directory. No portion of a data directory should be shared between multiple IcePatch servers. The data directory must contain only regular files and directories; it is not possible to transfer special files such as symbolic links or device files.

The `IcePatch.UpdatePeriod` property specifies the number of seconds between the server's periodic updates to its data directory. During an update, the server descends the data directory and performs a number of tasks, including compressing files and computing unique signatures. If not specified, the default value is 60. A value of 0 disables periodic updates.

The `IcePatch.BusyTimeout` property defines the number of seconds a client request should wait for access to a file or directory. Client requests may be delayed when the server is busy compressing a file or computing a unique signature. If the server remains busy after the specified timeout period, an `IcePatch::BusyException` is returned to the client. If not specified, the default value is 10.

The `IcePatch.Trace.Files` property determines whether the server emits diagnostic information to its configured logger. A value of 1 enables logging, while a value of 0 disables the output.

28.3.2 Starting the Server

Assuming the configuration properties shown in Section 28.3.1 are stored in a file named `config`, the server can be started with the following command:

```
$ icepatchserver --Ice.Config=config
```

The server can also be run as a Unix daemon or Win32 service. See Section 10.3.2 for more information.

28.3.3 Managing the Data Directory

The IcePatch server is designed to allow dynamic changes to its data directory, therefore it is not necessary to stop the server while the data directory is modified. Adding files or directories poses no problems whatsoever: if periodic updates are enabled (see Section 28.3.1), the server eventually compresses all of the new files and computes unique signatures. If a client requests a new file before the server has had a chance to compress it, the file is compressed immediately.

Replacing a file is also handled gracefully, even if a client is in the process of downloading the old file. In this case, the client continues to download the compressed version of the old file. Once the transfer completes, the server eventually compresses the new file when it detects that the file has a newer timestamp than its compressed version. Again, this occurs during the server's periodic updates, or when the file is next requested, whichever comes first.

Removing files or directories while the server is running can result in client failures, but does not harm the operational integrity of the server.

Note that it is possible for a client to receive an inconsistent set of files if it is transferring at the same time the server's data directory is being modified. If this scenario is undesirable, there are some strategies you can use to address it:

1. Stop the server before modifying the data directory. Although client transfers may be interrupted by stopping the server, the clients should receive a consistent file set when they reconnect upon server restart.
2. Deploy a new IcePatch server on a new endpoint to distribute the modified data directory. This strategy allows active client transfers to complete using the old server, but requires that clients have the ability to discover the new server.

28.3.4 Special Files

The IcePatch server does not create a central database to track information about the files and directories in its data directory. Rather, the server creates one or more

additional files for each file and directory. These special files have reserved extensions and are invisible to clients. Table 28.1 describes each of the special files.

Table 28.1. Special files.

File	Description
<i>file</i> .md5	The unique signature of <i>file</i> .
<i>file</i> .bz2	The compressed version of <i>file</i> .
<i>directory</i> .md5	The unique signature of <i>directory</i> .
<i>directory</i> .tot	The totals calculated for <i>directory</i> . See Section 28.6.

A file containing a unique signature has the .md5 extension because the signature is computed using the MD5 message digest algorithm commonly used in cryptographic applications. Similarly, the extension for compressed files is .bz2 because IcePatch uses the bzip2 compression algorithm.

A unique signature is also computed for the top-level data directory. Therefore, if the data directory is /home/icepatch/data, then the server also creates /home/icepatch/data.md5.

It is important to remember that any file in the data directory with an extension of .md5, .bz2 or .tot is invisible to clients. Data files and directories must not be given names with these reserved extensions.

28.4 The IcePatch Client

This section describes the configuration and use of the IcePatch client.

28.4.1 Configuring the Client

The IcePatch client supports a number of configuration properties, as described in Appendix C. A minimal configuration is shown below:

```
IcePatch.Endpoints=tcp -p 10000
IcePatch.Directory=localdata
IcePatch.RemoveOrphaned=1
IcePatch.Thorough=1
```

The `IcePatch.Endpoints` property specifies one or more endpoints on which the IcePatch server is listening for requests. This is the only mandatory property.

The `IcePatch.Directory` property specifies the client's local data directory. If not defined, the client uses its current working directory.

The `IcePatch.RemoveOrphaned` property determines whether the client automatically removes files from the local data directory that are no longer present in the server's data directory. If not specified, the default value is 0, meaning orphaned files are not removed.

The `IcePatch.Thorough` property determines the client's behavior when patching directories. A value greater than zero causes the client to compare the contents of a directory even when the directory's unique signature matches that of the server. If not specified, the default value is 0.

28.4.2 Glacier Support

The IcePatch client can be configured to use a Glacier starter by defining the `Glacier.Starter.Endpoints` property. For more information on the Glacier starter, see Section 25.5.

28.4.3 Starting the Client

Assuming the configuration properties shown in Section 28.4.1 are stored in a file named `config`, the client can be started with the following command:

```
$ icepatchclient --Ice.Config=config [DIR DIR ...]
```

If no directory arguments are provided, the client patches the local data directory against the server's top-level data directory. Otherwise, the directory arguments should be relative pathnames representing subdirectories of the top-level data directory. The server rejects requests for absolute pathnames, as well as pathnames containing parent directory (..) path components.

28.5 Developing an IcePatch Client

The `icepatchclient` command-line utility described in the Section 28.4 is easy to use, but may not provide the tight integration needed by some applications. For example, a graphical application might need the ability to patch itself while providing useful feedback (such as progress bars) to the user about its patching activity. In a situation such as this, a C++ application can employ the IcePatch Slice interfaces, as well as a library of C++ convenience functions, to implement an IcePatch client.

28.5.1 Slice Overview

The classes and interfaces comprising the IcePatch Slice API are shown in Figure 28.1.

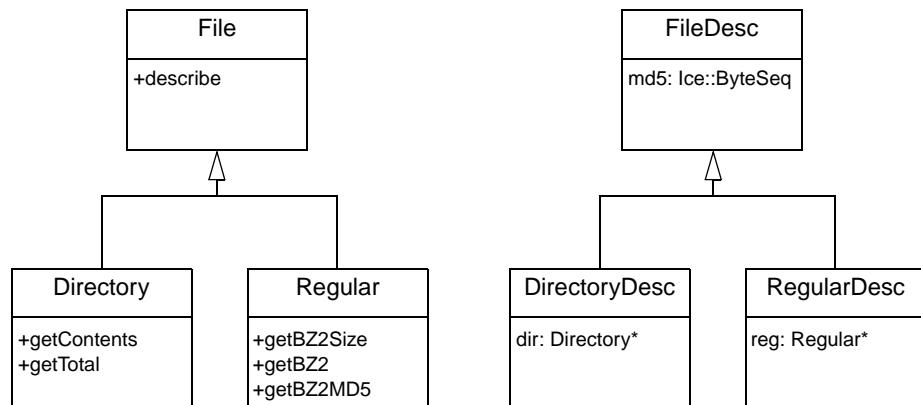


Figure 28.1. IcePatch Slice API

The interfaces `File`, `Directory` and `Regular` are accessed via proxies, whereas the classes `FileDesc`, `DirectoryDesc` and `RegularDesc` are passed by value.

File Operations

The `File` interface provides a single operation:

`nonmutating FileDesc describe()`

The return value of `describe` is a subclass of `FileDesc`, either `DirectoryDesc` or `RegularDesc`.

Directory Operations

The `Directory` interface provides the following operations:

```
nonmutating FileDescSeq getContents()  
nonmutating long getTotal(Ice::ByteSeq md5)
```

The `getContents` operation returns descriptions of the directory's immediate children. The `getTotal` operation returns the number of bytes to be downloaded in order to patch a local directory with the given unique signature to match the directory's current contents on the server (see Section 28.6).

Regular Operations

The `Regular` interface provides the following operations:

```
nonmutating int getBZ2Size()  
nonmutating Ice::ByteSeq getBZ2(int pos, int num)  
nonmutating Ice::ByteSeq getBZ2MD5(int size)
```

The `getBZ2Size` operation returns the size of the compressed file. The `getBZ2` operation returns a fragment of the compressed file starting at the position `pos` and containing at most `num` bytes.

The `getBZ2MD5` operation computes a unique signature for the first `size` bytes of the compressed file, and is used when the client finds a partially downloaded compressed file in the local data directory. In this situation, the client may try to preserve the partial file and resume the transfer, but can only do so if the contents of the partial file matches that on the server. The client therefore computes a unique signature for the partial file and asks the server to do the same. If the signatures match, the client can safely resume the transfer.

Description Classes

The `FileDesc` class contains an `md5` member representing the unique signature of the file or directory. The `DirectoryDesc` and `RegularDesc` subclasses contain a proxy for the directory and file, respectively.

28.5.2 Proxy Conventions

In order to construct a proxy for a `Directory` or `Regular` object, an `IcePatch` client must adhere to the conventions required by the `IcePatch` server for proxy

identities. In particular, the identity's category must be `IcePatch`, and the identity's name must be the pathname of the directory or file, relative to the top-level data directory. The pathname must always use Unix-style forward slashes (e.g., `subdir1/subdir2`). The identity name of the top-level directory is `.` (a period), as shown in the following example:

```
Ice::PropertiesPtr properties = communicator->getProperties();
string endpoints = properties->getProperty("IcePatch.Endpoints");
Ice::Identity id;
id.category = "IcePatch";
id.name = ".";
string str = Ice::identityToString(id) + ":" + endpoints;
IcePatch::DirectoryPrx top = IcePatch::DirectoryPrx::checkedCast(
    communicator->stringToProxy(str));
```

The client generally needs to bootstrap only the top-level directory in this fashion, since the proxies for files and subdirectories can be obtained from the descriptions returned by `Directory::getContents`.

Notice that the description classes do not contain a member representing the name of the directory or file. Such a member would be redundant, because the pathname is contained in the description's proxy member. The code below is a continuation of the previous example, illustrating how to display the name of each file and subdirectory of the top-level directory.

```
IcePatch::FileDescSeq contents = top->getContents();
for (IcePatch::FileDescSeq::iterator p = contents.begin();
     p != contents.end(); ++p)
{
    IcePatch::DirectoryDescPtr d =
        IcePatch::DirectoryDescPtr::dynamicCast(*p);
    if (d) {
        cout << d->dir->ice_getIdentity().name << endl;
    } else {
        IcePatch::RegularDescPtr r =
            IcePatch::RegularDescPtr::dynamicCast(*p);
        assert(r);
        cout << r->reg->ice_getIdentity().name << endl;
    }
}
```

It is important to remember however that the identity's name is a pathname, and therefore may contain leading subdirectories.

28.5.3 Convenience Functions

IcePatch includes a number of C++ convenience functions to aid in the development of an IcePatch client. The functions are defined in two header files: `IcePatch/ClientUtil.h` and `IcePatch/Util.h`. The functions in `IcePatch/ClientUtil.h` are useful only to IcePatch clients, whereas the functions in `IcePatch/Util.h` are used by both clients and the IcePatch server.

Client Functions

The header file `IcePatch/ClientUtil.h` contains two functions:

```
namespace IcePatch {
    std::string pathToName(const std::string& path);
    void getRegular(const IcePatch::RegularPrx& file,
                    ProgressCB& callback);
}
```

The `pathToName` function returns the last component of a pathname, which is useful when operating on the proxy identity name as shown in Section 28.5.2.

The `getRegular` function performs all of the steps necessary to transfer and uncompress a file. The function first checks the local data directory for the presence of a partially-downloaded compressed file, and attempts to resume that transfer if possible. Otherwise, the entire compressed file is transferred. Once the transfer is complete, the file is uncompressed. Any errors that occur are returned as exceptions, which may include Ice exceptions raised by remote invocations on the IcePatch server. For this reason, the client must be prepared to catch Ice and IcePatch exceptions.

The `ProgressCB` argument is an callback object invoked by `getRegular` to report its progress, and is defined as follows:

```
namespace IcePatch {
    class ProgressCB {
    public:
        virtual void startDownload(Ice::Int total,
                                   Ice::Int pos) = 0;
        virtual void updateDownload(Ice::Int total,
                                   Ice::Int pos) = 0;
        virtual void finishedDownload(Ice::Int total) = 0;

        virtual void startUncompress(Ice::Int total,
                                   Ice::Int pos) = 0;
        virtual void updateUncompress(Ice::Int total,
```

```
        Ice::Int pos) = 0;
    virtual void finishedUncompress(Ice::Int total) = 0;
};

}
```

The `startDownload` member function is called once the file transfer has begun. The integer arguments represent the total size of the compressed file and the starting position.

The `updateDownload` member function is called each time another fragment of the file has been transferred. The integer arguments represent the total size of the compressed file and the current position, including the most recent fragment.

The `finishedDownload` member function is called when the transfer completes successfully. The integer argument represents the total size of the compressed file.

The `startUncompress` member function is called when the uncompress stage has begun. The integer arguments represent the total size of the compressed file and the starting position (always zero).

The `updateUncompress` member function is called each time another fragment of the file has been uncompressed. The integer arguments represent the total size of the compressed file and the current position, including the most recent fragment.

The `finishedUncompress` member function is called when the uncompress stage completes successfully. The integer argument represents the total size of the compressed file.

Identity Functions

The header file `IcePatch/Util.h` contains two functions for converting the contents of an `Ice::Identity` value to and from a pathname:

```
namespace IcePatch {
    std::string identityToPath(const Ice::Identity& id);
    Ice::Identity pathToIdentity(const std::string& path);
}
```

The `identityToPath` function returns a normalized version of the identity's name, and the `pathToIdentity` function returns an identity containing the proper category and a normalized version of the given pathname. A normalized pathname is one in which all backslashes are converted to forward slashes, and any redundant pathname components are eliminated.

Suffix Functions

The header file `IcePatch/Util.h` contains three functions for operating on pathname suffixes:

```
namespace IcePatch {
    std::string getSuffix(const std::string& path);
    bool ignoreSuffix(const std::string& path);
    std::string removeSuffix(const std::string& path);
}
```

The `getSuffix` function returns the suffix of the given pathname, or an empty string if no suffix is found. The `ignoreSuffix` function returns true if the given pathname contains a suffix that is ignored by IcePatch. These suffixes include `.md5`, `.bz2` and `.tot`, as well as suffixes used for temporary files during file transfer and uncompression. The `removeSuffix` function returns the given pathname with its suffix removed.

File Attribute Functions

The header file `IcePatch/Util.h` contains one function for retrieving file system attributes:

```
namespace IcePatch {
    enum FileType {
        FileTypeNotExist,
        FileTypeDirectory,
        FileTypeRegular
    };

    struct FileInfo {
        FileType type;
        off_t size;
        time_t time;
    };

    FileInfo getFileInfo(const std::string& path,
                        bool exceptionIfNotExist,
                        const Ice::LoggerPtr& logger = 0);
}
```

The first argument to `getFileInfo` is the pathname of the local file or directory. The second argument is a flag indicating whether the function should raise an exception if the pathname does not exist. The final argument is an optional `Ice::Logger` object to which trace information may be sent.

If the boolean argument is true, an `IcePatch::FileAccessException` is raised if the pathname does not exist. If the boolean argument is true, the type member of the returned `FileInfo` value is set to `FileTypeNotExist`.

File Removal Functions

The header file `IcePatch/Util.h` contains one function for removing files and directories:

```
namespace IcePatch {
    void removeRecursive(const std::string& path,
                         const Ice::LoggerPtr& logger = 0);
}
```

The first argument to `removeRecursive` is the pathname of the local file or directory to be removed. The second argument is an optional `Ice::Logger` object to which trace information may be sent.

Directory Functions

The header file `IcePatch/Util.h` contains three functions for operating on filesystem directories:

```
namespace IcePatch {
    Ice::StringSeq readDirectory(const std::string& dir);
    Ice::StringSeq readDirectory(const std::string& dir,
                                const std::string& prefix);
    void createDirectory(const std::string& dir);
}
```

Two versions of the `readDirectory` function are provided. Both versions return the pathnames of the files and subdirectories contained in the given directory, but they differ in the format of the pathnames they return. The version taking two string arguments prepends the pathname of each file and subdirectory with the given directory prefix, whereas the version taking one string argument prepends each pathname with the directory being read.

The `createDirectory` function creates a new subdirectory with the given pathname. Intermediate subdirectories must already exist.

MD5 Functions

The header file `IcePatch/Util.h` contains several functions for computing unique signatures using the MD5 algorithm and storing them in files:

```
namespace IcePatch {
    Ice::ByteSeq getMD5(const std::string& path);
    void putMD5(const std::string& path, const Ice::ByteSeq& md5);
    void createMD5(const std::string& path,
                    const Ice::LoggerPtr& logger = 0);
    Ice::ByteSeq calcMD5(const std::string& path,
                          const Ice::LoggerPtr& logger = 0);
    Ice::ByteSeq calcPartialMD5(const std::string& file,
                                Ice::Int size,
                                const Ice::LoggerPtr& logger = 0);
}
```

All functions that operate on an MD5 file append the suffix .md5 to the given pathname. An MD5 file contains 16 bytes representing the unique signature of a file or subdirectory.

The `getMD5` function returns the contents of the MD5 file for the file or subdirectory identified by the given pathname. The function returns `IcePatch::FileAccessException` if the MD5 file does not exist or cannot be read.

The `putMD5` function stores the given unique signature into an MD5 file for the given file or subdirectory. The function returns `IcePatch::FileAccessException` if the MD5 file cannot be created.

The `createMD5` function calculates the unique signature of the given file or subdirectory and stores it in an MD5 file. The second argument is an optional `Ice::Logger` object to which trace information may be sent.

The `calcMD5` function returns the unique signature of the given file or subdirectory. The second argument is an optional `Ice::Logger` object to which trace information may be sent.

The `calcPartialMD5` function returns the unique signature of a portion of the given file. The `size` argument specifies the number of bytes for which to compute the signature. The second argument is an optional `Ice::Logger` object to which trace information may be sent. The function returns `IcePatch::FileAccessException` if the file does not exist or does not contain at least `size` bytes.

Compression Functions

The header file `IcePatch/Util.h` contains two functions for manipulating compressed files:

```
namespace IcePatch {
    Ice::ByteSeq getBZ2(const std::string& path, Ice::Int pos,
                         Ice::Int num);
    void createBZ2(const std::string& path,
                    const Ice::LoggerPtr& logger);
}
```

The `getBZ2` function returns at most `num` bytes of the compressed file associated with the given pathname. The function appends the suffix `.bz2` to the pathname. The function returns `IcePatch::FileAccessException` if the MD5 file cannot be opened or read.

The `createBZ2` function creates a compressed file associated with the given pathname. The function creates a temporary file during compression with the suffix `.bz2temp`, and renames this file to have a `.bz2` suffix after compression succeeds.

28.6 Download Totals

The callback mechanism described in Section 28.5.3 notifies the client about the progress of each file that is downloaded and uncompressed. However, the IcePatch server’s default configuration provides no information to the client about its overall progress during the patching session. Because the IcePatch server allows dynamic updates to its data directory, any estimate the server might offer on the total number of bytes a client must download during the session would be questionable at best, and potentially quite inaccurate.

28.6.1 `icepatchcalc`

For situations in which an IcePatch client must be able to reliably determine its progress during patching, IcePatch provides the `icepatchcalc` tool. In order to take advantage of this tool, the IcePatch server’s data directory must be updated in discrete “releases.” The `icepatchcalc` tool calculates the size of each subdirectory, and optionally compares the new release against one or more prior releases to determine the number of bytes a client possessing a prior release would need to download in order to match the new release. The totals are computed for the top-level data directory, as well as for each subdirectory, and stored in the server’s data directory in special files having a `.tot` suffix.

For example, suppose the first release of our IcePatch server's data directory is /opt/icepatch/data/V1_0. We run `icepatchcalc` on this directory to calculate the download totals:

```
% icepatchcalc /opt/icepatch/data/V1_0
```

The tool traverses the entire data directory to ensure that every file is compressed and that all unique signatures are updated. Next, it calculates the size of each subdirectory and stores the total in a special file having the name of the subdirectory with a `.tot` suffix. In our example, the total for the top-level data directory is stored in the file /opt/icepatch/data/V1_0.tot.

When we are ready to release a new version of the data directory, we follow these steps:

1. Create the new data directory (e.g., /opt/icepatch/data/V1_1).
2. Populate the new data directory.
3. Run `icepatchcalc` on the new data directory. We also supply the path-name of the prior release's data directory in order to calculate the download totals for clients that already possess the previous version:

```
% icepatchcalc /opt/icepatch/data/V1_1 \
  /opt/icepatch/data/V1_0
```

4. Stop the IcePatch server.
5. Modify the IcePatch server configuration to use the new data directory.
6. Start the IcePatch server.

Having completed this preparation, an IcePatch client can obtain the total number of bytes to be transferred when it is downloading the entire data directory for a new installation, and when patching an installation of V1_0.

Note: *Use caution when preparing a new data directory, especially when copying files from the data directory of a prior release. It is imperative that no IcePatch special files be copied to the new data directory.*

28.6.2 Details

Unique signatures are the critical factor in calculating download totals. When an IcePatch server's data directory is managed in discrete releases, the signature computed for the top-level data directory uniquely identifies that release. The `icepatchcalc` tool works by building a table for each subdirectory that associates the unique signature of a previous release with the total number of bytes necessary to patch that release.

Furthermore, the client computes an identical signature for its local copy of the data directory. Therefore, an IcePatch client can supply the unique signature of its local data directory to the `getTotal` operation (see Section 28.5.1) to obtain the total number of bytes the client must download in order to patch its local data directory to match the server's. The client can also supply an empty signature to `getTotal` if it wishes to obtain the number of bytes in a complete transfer.

28.7 Summary

IcePatch addresses a requirement common to both development and deployment scenarios: the safe, secure, and efficient replication of a directory tree. The IcePatch server is easy to configure and is flexible enough to support dynamic additions to its data directory. For simple uses, IcePatch provides a client that can be used to patch directory hierarchies from the command line. The IcePatch API permits tight integration of applications with the IcePatch functionality. Finally, administrators have the option of delivering discrete IcePatch releases, with the advantage of enabling clients to readily determine their progress during a patching session.

Appendices

Appendix A

Slice Keywords

The following identifiers are Slice keywords:

bool	enum	implements	module	string
byte	exception	int	non mutating	struct
class	extends	interface	Object	throws
const	false	local	out	true
dictionary	float	LocalObject	sequence	void
double	idempotent	long	short	

Keywords must be capitalized as shown.

Appendix B

Slice Documentation

B.1 Ice

Overview

```
module Ice
```

The Ice core library. Among many other features, the Ice core library manages all the communication tasks using an efficient protocol (including protocol compression and support for both TCP and UDP), provides a thread pool for multi-threaded servers, and additional functionality that supports high scalability.

BoolSeq

```
[ "cs:array" ] sequence<bool> BoolSeq;
```

A sequence of bools.

ByteSeq

```
[ "cs:array" ] sequence<byte> ByteSeq;
```

A sequence of bytes.

Used By

```
::Glacier::Starter::startRouter, ::Glacier::Starter::startRouter,  
::Glacier::Starter::startRouter, BadMagicException::badMagic,  
::IcePatch::Directory::getTotal, ::IcePatch::FileDesc::md5,  
::IcePatch::Regular::getBZ2, ::IcePatch::Regular::getBZ2MD5,  
::IceSSL::Plugin::addTrustedCertificate, ::IceSSL::Plugin::getSingleCertVerifier,  
::IceSSL::Plugin::setRSAKeys,  
::IceSSL::Plugin::setRSAKeys.
```

DoubleSeq

```
[ "cs:array" ] sequence<double> DoubleSeq;
```

A sequence of doubles.

FloatSeq

```
[ "cs:array" ] sequence<float> FloatSeq;
```

A sequence of floats.

IdentitySeq

```
sequence<Identity> IdentitySeq;
```

A sequence of identities.

IntSeq

```
[ "cs:array" ] sequence<int> IntSeq;
```

A sequence of ints.

LongSeq

```
[ "cs:array" ] sequence<long> LongSeq;
```

A sequence of longs.

ObjectProxySeq

```
sequence<Object*> ObjectProxySeq;
```

A sequence of object proxies.

Used By

```
::IcePack::Query::findAllObjectsWithType.
```

ObjectSeq

```
sequence<Object> ObjectSeq;
```

A sequence of objects.

ShortSeq

```
[ "cs:array" ] sequence<short> ShortSeq;
```

A sequence of shorts.

StringSeq

```
[ "cs:array" ] sequence<string> StringSeq;
```

A sequence of strings.

Used By

```
Properties::getCommandLineOptions, Properties::parseCommandLineOptions, Properties::parseCommandLineOptions, Properties::parseIceCommandLineOptions, Properties::parseIceCommandLineOptions, ::IceBox::FreezeService::start, ::IceBox::Service::start, ::IcePack::Admin::addApplication, ::IcePack::Admin::addServer, ::IcePack::Admin::getAllAdapterIds, ::IcePack::Admin::getAllNodeNames, ::IcePack::Admin::getAllServerNames, ::IcePack::ServerDescription::args, ::IcePack::ServerDescription::envs, ::IcePack::ServerDescription::targets.
```

Context

```
dictionary<string, string> Context;
```

A request context. `Context` is used to transmit metadata about a request from the server to the client, such as Quality-of-Service (QoS) parameters. Each operation on the client has a `Context` as its implicit final parameter.

Used By

`Current::ctx`.

FacetMap

```
local dictionary<string, Object> FacetMap;
```

A mapping from facet name to servant.

Used By

`ObjectAdapter::findAllFacets`, `ObjectAdapter::removeAllFacets`.

ObjectDict

```
local dictionary<Identity, Object> ObjectDict;
```

A mapping between identities and Ice objects.

PropertyDict

```
local dictionary<string, string> PropertyDict;
```

A simple collection of properties, represented as a dictionary of key/value pairs. Both key and value are `strings`.

Used By

`Properties::getPropertiesForPrefix`.

See Also

`Properties::getPropertiesForPrefix`.

B.2 Ice::AbortBatchRequestException

Overview

local exception AbortBatchRequestException extends
ProtocolException

This exception is a specialization of `ProtocolException`, indicating that a batch request has been aborted.

B.3 Ice::AdapterAlreadyActiveException

Overview

exception AdapterAlreadyActiveException

This exception is raised if a server tries to set endpoints for an adapter that is already active.

B.4 Ice::AdapterNotFoundException

Overview

exception AdapterNotFoundException

This exception is raised if an adapter cannot be found.

B.5 Ice::AlreadyRegisteredException

Overview

local exception AlreadyRegisteredException

This exception is raised if an attempt is made to register a servant, servant locator, facet, object factory, plug-in, object adapter, object, or user exception factory more than once for the same ID.

id

```
string id;
```

The id (or name) of the object that is registered already.

kindOfObject

```
string kindOfObject;
```

The kind of object that is registered already: "servant", "servant locator", "facet", "object factory", "plug-in", "object adapter", "object", or "user exception factory".

B.6 Ice::BadMagicException

Overview

```
local exception BadMagicException extends ProtocolException
```

This exception is a specialization of `ProtocolException`, indicating that a message did not start with the expected magic number ('I', 'c', 'e', 'P').

badMagic

```
ByteSeq badMagic;
```

A sequence containing the first four bytes of the incorrect message.

B.7 Ice::CloseConnectionException

Overview

```
local exception CloseConnectionException extends ProtocolException
```

This exception is a specialization of `ProtocolException`, indicating that the connection has been gracefully shut down by the server. The operation call that caused this exception has not been executed by the server. In most cases you will not get this exception, because the client will automatically retry the operation call in case the server shut down the connection. However, if upon retry the server shuts down the connection again, and the retry limit has been reached, then this exception is propagated to the application code.

B.8 `Ice::CloseTimeoutException`

Overview

```
local exception CloseTimeoutException extends TimeoutException
```

This exception is a specialization of `TimeoutException` for connection closure timeout conditions.

B.9 `Ice::CollocationOptimizationException`

Overview

```
local exception CollocationOptimizationException
```

This exception is raised if a feature is requested that is not supported with collocation optimization.

B.10 `Ice::Communicator`

Overview

```
local interface Communicator
```

The central object in Ice. One or more communicators can be instantiated for an Ice application. Communicator instantiation is language specific, and not specified in Slice code.

Used By

`::Freeze::Connection::getCommunicator, ObjectAdapter::getCommunicator, ::IceBox::FreezeService::start, ::IceBox::Service::start.`

See Also

`Logger, Stats, ObjectAdapter, Properties, ObjectFactory.`

`addObjectFactory`

```
void addObjectFactory(ObjectFactory factory, string id);
```

Add a servant factory to this communicator. Installing a factory with an id for which a factory is already registered throws `AlreadyRegisteredException`.

When unmarshaling an Ice object, the Ice run-time reads the most-derived type id off the wire and attempts to create an instance of the type using a factory. If no instance is created, either because no factory was found, or because all factories returned nil, the object is sliced to the next most-derived type and the process repeats. If no factory is found that can create an instance, the Ice run-time will slice the object to the type `Ice::Object`.

The following order is used to locate a factory for a type:

1. The Ice run-time looks for a factory registered specifically for the type.
2. If no instance has been created, the Ice run-time looks for the default factory, which is registered with an empty type id.
3. If no instance has been created by any of the preceding steps, the Ice run-time looks for a factory that may have been statically generated by the language mapping for non-abstract classes.

Parameters

`factory`

The factory to add.

`id`

The type id for which the factory can create instances, or an empty string for the default factory.

See Also

`removeObjectFactory, findObjectFactory, ObjectFactory.`

createObjectAdapter

```
ObjectAdapter createObjectAdapter(string name);
```

Create a new object adapter. The endpoints for the object adapter are taken from the property *name*.Endpoints.

Parameters

name

The object adapter name.

Return Value

The new object adapter.

See Also

[createObjectAdapterWithEndpoints](#), [ObjectAdapter](#), [Properties](#).

createObjectAdapterWithEndpoints

```
ObjectAdapter createObjectAdapterWithEndpoints(string name, string  
endpoints);
```

Create a new object adapter with endpoints. This method sets the property *name*.Endpoints, and then calls `createObjectAdapter`. It is provided as a convenience function.

Parameters

name

The object adapter name.

endpoints

The endpoints for the object adapter.

Return Value

The new object adapter.

See Also

[createObjectAdapter](#), [ObjectAdapter](#), [Properties](#).

destroy

```
void destroy();
```

Destroy the communicator. This operation calls `shutdown` implicitly. Calling `destroy` cleans up memory, and shuts down this communicator's client functionality. Subsequent calls to `destroy` are ignored.

See Also

`shutdown`.

findObjectFactory

```
ObjectFactory findObjectFactory(string id);
```

Find a servant factory registered with this communicator.

Parameters

`id`

The type id for which the factory can create instances, or an empty string for the default factory.

Return Value

The servant factory, or null if no servant factory was found for the given id.

See Also

`addObjectFactory`, `removeObjectFactory`, `ObjectFactory`.

flushBatchRequests

```
void flushBatchRequests();
```

Flush any pending batch requests for this communicator. This causes all batch requests that were sent via proxies obtained via this communicator to be sent to the server.

getLogger

```
Logger getLogger();
```

Get the logger for this communicator.

Return Value

This communicator's logger.

See Also

Logger.

getPluginManager

```
PluginManager getPluginManager();
```

Get the plug-in manager for this communicator.

Return Value

This communicator's plug-in manager.

See Also

PluginManager.

getProperties

```
Properties getProperties();
```

Get the properties for this communicator.

Return Value

This communicator's properties.

See Also

Properties.

getStats

```
Stats getStats();
```

Get the statistics callback object for this communicator.

Return Value

This communicator's statistics callback object.

See Also

[Stats](#).

proxyToString

```
string proxyToString(Object* obj);
```

Convert a proxy into a string.

Parameters

obj

The proxy to convert into a string.

Return Value

The "stringified" proxy.

See Also

[stringToProxy](#).

removeObjectFactory

```
void removeObjectFactory(string id);
```

Remove a servant factory from this communicator. Removing an id for which no factory is registered throws `NotRegisteredException`.

Parameters

id

The type id for which the factory can create instances, or an empty string for the default factory.

See Also

[addObjectFactory](#), [findObjectFactory](#), [ObjectFactory](#).

setDefaultLocator

```
void setDefaultLocator(Locator* loc);
```

Set a default Ice locator for this communicator. All newly created proxy and object adapters will use this default locator. To disable the default locator, null can be used. Note that this operation has no effect on existing proxies or object adapters.

You can also set a locator for an individual proxy by calling the operation `ice_locator` on the proxy, or for an object adapter by calling the operation `setLocator` on the object adapter.

Parameters

`loc`

The default locator to use for this communicator.

See Also

`Locator`, `ObjectAdapter::setLocator`.

setDefaultRouter

```
void setDefaultRouter(Router* rtr);
```

Set a default router for this communicator. All newly created proxies will use this default router. To disable the default router, null can be used. Note that this operation has no effect on existing proxies.

You can also set a router for an individual proxy by calling the operation `ice_router` on the proxy.

Parameters

`rtr`

The default router to use for this communicator.

See Also

`Router`, `ObjectAdapter::addRouter`.

setLogger

```
void setLogger(Logger log);
```

Set the logger for this communicator.

Parameters

`log`

The logger to use for this communicator.

See Also

`Logger`.

`setStats`

`void setStats(Stats st);`

Set the statistics callback object for this communicator.

Parameters

`st`

The statistics callback object to use for this communicator.

See Also

`Stats`.

`shutdown`

`void shutdown();`

Shuts down this communicator's server functionality, including the deactivation of all object adapters. Subsequent calls to `shutdown` are ignored.

After `shutdown` returns, no new requests are processed. However, requests that have been started before `shutdown` was called might still be active. You can use `waitForShutdown` to wait for the completion of all requests.

See Also

`destroy`, `waitForShutdown`, `ObjectAdapter::deactivate`.

`stringToProxy`

`Object* stringToProxy(string str);`

Convert a string into a proxy. For example, `MyCategory/MyObject:tcp -h some_host -p 10000` creates a proxy that refers to the Ice object having an identity with a name "MyObject" and a category "MyCategory", with the server running on host "some_host", port 10000.

Parameters

`str`

The string to convert into a proxy.

Return Value

The proxy.

See Also

`proxyToString`.

`waitForShutdown`

`void waitForShutdown();`

Wait until this communicator's server functionality has shut down completely. Calling `shutdown` initiates shutdown, and `waitForShutdown` only returns when all outstanding requests have completed. A typical use of this operation is to call it from the main thread, which then waits until some other thread calls `shutdown`. After shutdown is complete, the main thread returns and can do some cleanup work before it finally calls `destroy` to also shut down the client functionality, and then exits the application.

See Also

`shutdown, destroy, ObjectAdapter::waitForDeactivate`.

B.11 `Ice::CommunicatorDestroyedException`

Overview

`local exception CommunicatorDestroyedException`

This exception is raised if the `Communicator` has been destroyed.

See Also

`Communicator::destroy.`

B.12 `Ice::CompressionException`

Overview

`local exception CompressionException extends ProtocolException`

This exception is a specialization of `ProtocolException` that is raised if there is a problem with compressing or uncompressing data.

`reason`

`string reason;`

A description of the problem with compress or uncompress.

B.13 `Ice::CompressionNotSupportedException`

Overview

`local exception CompressionNotSupportedException extends ProtocolException`

This exception is a specialization of `ProtocolException` that is raised if a compressed protocol message has been received by an Ice version that does not support compression.

B.14 `Ice::ConnectFailedException`

Overview

`local exception ConnectFailedException extends SocketException`

This exception is a specialization of `SocketException` for connection failures.

Derived Exceptions

`ConnectionRefusedException`.

B.15 `Ice::ConnectTimeoutException`

Overview

Local exception `ConnectTimeoutException` extends `TimeoutException`

This exception is a specialization of `TimeoutException` for connection establishment timeout conditions.

B.16 `Ice::ConnectionLostException`

Overview

Local exception `ConnectionLostException` extends `SocketException`

This exception is a specialization of `SocketException`, indicating a lost connection.

B.17 `Ice::ConnectionNotValidatedException`

Overview

Local exception `ConnectionNotValidatedException` extends `ProtocolException`

This exception is a specialization of `ProtocolException`, that is raised if a message is received over a connection that is not yet validated.

B.18 `Ice::ConnectionRefusedException`

Overview

```
local exception ConnectionRefusedException extends  
ConnectFailedException
```

This exception is a specialization of `ConnectFailedException` for connection failures, where the server host actively refuses a connection.

B.19 `Ice::ConnectionTimeoutException`

Overview

```
local exception ConnectionTimeoutException extends  
TimeoutException
```

This exception is a specialization of `TimeoutException`, and indicates that a connection has been shut down because it has been idle for some time.

B.20 `Ice::Current`

Overview

```
local struct Current
```

Information about the current method invocation for servers. Each operation on the server has a `Current` as its implicit final parameter. `Current` is mostly used for Ice services, such as IceStorm. Most applications ignore this parameter.

Used By

`ServantLocator::finished`, `ServantLocator::locate`.

See Also

`::IceStorm`.

adapter

```
ObjectAdapter adapter;
```

The object adapter.

ctx

```
Context ctx;
```

The request context, as received from the client.

facet

```
string facet;
```

The facet.

id

```
Identity id;
```

The Ice object identity.

mode

```
OperationMode mode;
```

The mode of the operation.

operation

```
string operation;
```

The operation name.

B.21 Ice::DNSException

Overview

```
local exception DNSException
```

This exception indicates a DNS problem. For details on the cause, `error` should be inspected.

`error`

```
int error;
```

The error number describing the DNS problem. For C++ and Unix, this is equivalent to `h_errno`. For C++ and Windows, this is the value returned by `WSAGetLastError()`.

`host`

```
string host;
```

The host name that could not be resolved.

B.22 Ice::DatagramLimitException

Overview

```
local exception DatagramLimitException extends ProtocolException
```

This exception is a specialization of `ProtocolException` that is raised if a datagram exceeds the configured send or receive buffer size, or exceeds the maximum payload size of a UDP packet (65507 bytes).

B.23 `Ice::EncapsulationException`

Overview

```
local exception EncapsulationException extends MarshalException
```

This exception is a specialization of `MarshalException`, indicating a malformed data encapsulation.

B.24 `Ice::EndpointParseException`

Overview

```
local exception EndpointParseException
```

This exception is raised if there was an error while parsing an endpoint.

```
str
```

```
string str;
```

The string that could not be parsed.

B.25 `Ice::FacetNotExistException`

Overview

```
local exception FacetNotExistException extends  
RequestFailedException
```

This exception is raised if no facet with the given name exists, but at least one facet with the given identity exists.

B.26 Ice::Identity

Overview

```
struct Identity
```

The identity of an Ice object. An empty name denotes a null object.

Used By

```
::Freeze::Evictor::add, ::Freeze::Evictor::addFacet,  
::Freeze::Evictor::createObject, ::Freeze::Evictor::destroyObject,  
::Freeze::Evictor::hasFacet, ::Freeze::Evictor::hasObject,  
::Freeze::Evictor::keep, ::Freeze::Evictor::keepFacet,  
::Freeze::Evictor::release, ::Freeze::Evictor::releaseFacet,  
::Freeze::Evictor::remove, ::Freeze::Evictor::removeFacet,  
::Freeze::EvictorIterator::next,  
::Freeze::ServantInitializer::initialize, Current::id, IdentitySeq,  
IllegalIdentityException::id, Locator::findObjectById,  
ObjectAdapter::add, ObjectAdapter::addFacet, ObjectAdapter::create-  
DirectProxy, ObjectAdapter::createProxy, ObjectAdapter::createRe-  
verseProxy, ObjectAdapter::find, ObjectAdapter::findAllFacets,  
ObjectAdapter::findFacet, ObjectAdapter::remove,  
ObjectAdapter::removeAllFacets, ObjectAdapter::removeFacet, Object-  
Dict, RequestFailedException::id, ::IcePack::Query::findObjectById.
```

category

```
string category;
```

The Ice object category.

See Also

ServantLocator, ObjectAdapter::addServantLocator.

name

```
string name;
```

The name of the Ice object.

B.27 `Ice::IdentityParseException`

Overview

```
local exception IdentityParseException
```

This exception is raised if there was an error while parsing a stringified identity.

`str`

```
string str;
```

The string that could not be parsed.

B.28 `Ice::IllegalIdentityException`

Overview

```
local exception IllegalIdentityException
```

This exception is raised if an illegal identity is encountered.

`id`

```
Identity id;
```

The illegal identity.

B.29 `Ice::IllegalIndirectionException`

Overview

```
local exception IllegalIndirectionException extends  
MarshalException
```

This exception is a specialization of `MarshalException`, indicating an illegal indirection during unmarshaling.

B.30 `Ice::IllegalMessageSizeException`

Overview

```
local exception IllegalMessageSizeException extends  
ProtocolException
```

This exception is a specialization of `ProtocolException`, indicating that the message size is illegal, i.e., it is less than the minimum required size.

B.31 `Ice::Locator`

Overview

```
interface Locator
```

The Ice locator interface. This interface is used by clients to lookup adapters and objects. It is also used by servers to get the locator registry proxy.

The `Locator` interface is intended to be used by Ice internals and by locator implementations. Regular user code should not attempt to use any functionality of this interface directly.

`findAdapterById`

```
[ "amd" ] Object* findAdapterById(string id) throws  
AdapterNotFoundException;
```

Find an adapter by id and return its proxy (a dummy direct proxy created by the adapter).

Parameters

`id`

The adapter id.

Return Value

The adapter proxy, or null if the adapter is not active.

Exceptions**AdapterNotFoundException**

Raised if the adapter cannot be found.

findObjectById

```
[ "amd" ] Object* findObjectById(Identity id) throws  
ObjectNotFoundException;
```

Find an object by identity and return its proxy.

Parameters**id**

The identity.

Return Value

The proxy, or null if the object is not active.

Exceptions**ObjectNotFoundException**

Raised if the object cannot be found.

getRegistry

```
LocatorRegistry* getRegistry();
```

Get the locator registry.

Return Value

The locator registry.

B.32 **Ice::LocatorRegistry**

Overview

```
interface LocatorRegistry
```

The Ice locator registry interface. This interface is used by servers to register adapter endpoints with the locator.

The `LocatorRegistry` interface is intended to be used by Ice internals and by locator implementations. Regular user code should not attempt to use any functionality of this interface directly.

`setAdapterDirectProxy`

```
void setAdapterDirectProxy(string id, Object* proxy) throws  
AdapterNotFoundException, AdapterAlreadyActiveException;
```

Set the adapter endpoints with the locator registry.

Parameters

`id`

The adapter id.

`proxy`

The adapter proxy (a dummy direct proxy created by the adapter). The direct proxy contains the adapter endpoints.

Exceptions

`AdapterNotFoundException`

Raised if the adapter cannot be found, or if the locator only allows registered adapters to set their active proxy and the adapter is not registered with the locator.

`AdapterAlreadyActive`

Raised if an adapter with the same id is already active.

`setServerProcessProxy`

```
void setServerProcessProxy(string id, Process* proxy) throws  
ServerNotFoundException;
```

Set the process proxy for a server.

Parameters**id**

The server id.

proxy

The process proxy.

Exceptions**ServerNotFoundException**

Raised if the server cannot be found.

B.33 `Ice::Logger`

Overview**local interface Logger**

The Ice message logger. Applications can provide their own logger by implementing this interface and installing it in a communicator.

Used By**Communicator::getLogger, Communicator::setLogger.****error****void error(string message);**

Log error messages.

Parameters**message**

The error message to log.

See Also**warning.**

trace

```
void trace(string category, string message);
```

Log trace messages.

Parameters**category**

The trace category.

message

The trace message to log.

warning

```
void warning(string message);
```

Log warning messages.

Parameters**message**

The warning message to log.

See Also

`error`.

B.34 [Ice::MarshalException](#)**Overview**

```
local exception MarshalException extends ProtocolException
```

This exception is a specialization of `ProtocolException` that is raised upon an error during marshaling or unmarshaling data.

Derived Exceptions

EncapsulationException, IllegalIndirectionException, MemoryLimitException, NegativeSizeException, NoObjectFactoryException, ProxyUnmarshalException, UnmarshalOutOfBoundsException.

B.35 Ice::MemoryLimitException

Overview

local exception MemoryLimitException extends MarshalException

This exception is a specialization of MarshalException that is raised if the system-specific memory limit is exceeded during marshaling or unmarshaling.

B.36 Ice::NegativeSizeException

Overview

local exception NegativeSizeException extends MarshalException

This exception is a specialization of MarshalException that is raised if a negative size (e.g., a negative sequence size) is received.

B.37 Ice::NoEndpointException

Overview

local exception NoEndpointException

This exception is raised if no suitable endpoint is available.

```
proxy  
string proxy;
```

The stringified proxy for which no suitable endpoint is available.

B.38 Ice::NoObjectFactoryException

Overview

```
local exception NoObjectFactoryException extends MarshalException
```

This exception is a specialization of `MarshalException` that is raised if no suitable object factory was found during object unmarshaling.

See Also

`ObjectFactory`, `Communicator::addObjectFactory`, `Communicator::removeObjectFactory`, `Communicator::findObjectFactory`.

`type`

```
string type;
```

The absolute Slice type name of the object for which we could not find a factory.

B.39 Ice::NotRegisteredException

Overview

```
local exception NotRegisteredException
```

This exception is raised if an attempt is made to remove a servant, facet, object factory, plug-in, object adapter, object, or user exception factory that is not currently registered.

`id`

```
string id;
```

The id (or name) of the object that could not be removed.

```
kindOfObject  
string kindOfObject;
```

The kind of object that could not be removed: "servant", "facet", "object factory", "plug-in", "object adapter", "object", or "user exception factory".

B.40 `Ice::ObjectAdapter`

Overview

```
local interface ObjectAdapter
```

The object adapter, which is responsible for receiving requests from endpoints, and for mapping between servants, identities, and proxies.

Used By

```
::Freeze::ServantInitializer::initialize, Communicator::createObjectAdapter, Communicator::createObjectAdapterWithEndpoints, Current::adapter.
```

See Also

`Communicator`, `ServantLocator`.

activate

```
void activate();
```

Activate all endpoints that belong to this object adapter. After activation, the object adapter can dispatch requests received through its endpoints.

See Also

`hold`, `deactivate`.

add

```
Object* add(Object servant, Identity id);
```

Add a servant to this object adapter's Active Servant Map. Note that one servant can implement several Ice objects by registering the servant with multiple identities. Adding a servant with an identity that is in the map already throws `AlreadyRegisteredException`.

Parameters

`servant`

The servant to add.

`id`

The identity of the Ice object that is implemented by the servant.

Return Value

A proxy that matches the given identity and this object adapter.

See Also

`Identity`, `addFacet`, `addWithUUID`, `remove`, `find`.

addFacet

```
Object* addFacet(Object servant, Identity id, string facet);
```

Like `add`, but with a facet. Calling `add(servant, * id)` is equivalent to calling `addFacet` with an empty facet.

Parameters

`servant`

The servant to add.

`id`

The identity of the Ice object that is implemented by the servant.

`facet`

The facet. An empty facet means the default facet.

Return Value

A proxy that matches the given identity, facet, and this object adapter.

See Also

`Identity, add, addFacetWithUUID, removeFacet, findFacet.`

addFacetWithUUID

```
Object* addFacetWithUUID(Object servant, string facet);
```

Like `addWithUUID`, but with a facet. Calling `addWithUUID(servant)` is equivalent to calling `addFacetWithUUID` with an empty facet.

Parameters

`servant`

The servant to add.

`facet`

The facet. An empty facet means the default facet.

Return Value

A proxy that matches the generated UUID identity, facet, and this object adapter.

See Also

`Identity, addFacet, addWithUUID, removeFacet, findFacet.`

addRouter

```
void addRouter(Router* rtr);
```

Add a router to this object adapter. By doing so, this object adapter can receive callbacks from this router over connections that are established from this process to the router. This avoids the need for the router to establish a separate connection back to this object adapter.

You can add a particular router to only a single object adapter. Adding the same router to more than one object adapter results in undefined behavior. However, it is possible to add different routers to different object adapters.

Parameters

`rtr`

The router to add to this object adapter.

See Also

`Router, Communicator::setDefaultRouter.`

addServantLocator

```
void addServantLocator(ServantLocator locator, string category);
```

Add a Servant Locator to this object adapter. Adding a servant locator for a category for which a servant locator is already registered throws `AlreadyRegisteredException`. To dispatch operation calls on servants, the object adapter tries to find a servant for a given Ice object identity and facet in the following order:

1. The object adapter tries to find a servant for the identity and facet in the Active Servant Map.
2. If no servant has been found in the Active Servant Map, the object adapter tries to find a locator for the category component of the identity. If a locator is found, the object adapter tries to find a servant using this locator.
3. If no servant has been found by any of the preceding steps, the object adapter tries to find a locator for an empty category, regardless of the category contained in the identity. If a locator is found, the object adapter tries to find a servant using this locator.
4. If no servant has been found with any of the preceding steps, the object adapter gives up and the caller receives `ObjectNotExistException` or `FacetNotExistException`.

Only one locator for the empty category can be installed.

Parameters**locator**

The locator to add.

category

The category for which the Servant Locator can locate servants, or an empty string if the Servant Locator does not belong to any specific category.

See Also

`Identity, findServantLocator, ServantLocator.`

addWithUUID

```
Object* addWithUUID(Object servant);
```

Add a servant to this object adapter's Active Servant Map, using an automatically generated UUID as its identity. Note that the generated UUID identity can be accessed using the proxy's `ice_getIdentity` operation.

Parameters

servant

The servant to add.

Return Value

A proxy that matches the generated UUID identity and this object adapter.

See Also

`Identity`, `add`, `addFacetWithUUID`, `remove`, `find`.

createDirectProxy

```
Object* createDirectProxy(Identity id);
```

Create a "direct proxy" that matches this object adapter and the given identity. A direct proxy always contains the current adapter endpoints.

This operation is intended to be used by locator implementations. Regular user code should not attempt to use this operation.

Parameters

id

The identity for which a proxy is to be created.

Return Value

A proxy that matches the given identity and this object adapter.

See Also

`Identity`.

createProxy

```
Object* createProxy(Identity id);
```

Create a proxy that matches this object adapter and the given identity.

Parameters

id

The identity for which a proxy is to be created.

Return Value

A proxy that matches the given identity and this object adapter.

See Also

[Identity](#).

createReverseProxy

```
Object* createReverseProxy(Identity id);
```

Create a "reverse proxy" that matches this object adapter and the given identity. A reverse proxy uses connections that have been established from a client to this object adapter.

Like the [Router](#) interface, this operation is intended to be used by router implementations. Regular user code should not attempt to use this operation.

Parameters

id

The identity for which a proxy is to be created.

Return Value

A "reverse proxy" that matches the given identity and this object adapter.

See Also

[Identity](#).

deactivate

```
void deactivate();
```

Deactivate all endpoints that belong to this object adapter. After deactivation, the object adapter stops receiving requests through its endpoints. Object adapters that have been deactivated must not be reactivated again, i.e., the deactivation is permanent and `activate` or `hold` must not be called after calling `deactivate`; attempting to do so results in an `ObjectAdapterDeactivatedException` being thrown. Calls to `deactivate` on an already deactivated object adapter are ignored. After `deactivate` returns, no new requests are processed by the object adapter. However, requests that have been started before `deactivate` was called might still be active. You can use `waitForDeactivate` to wait for the completion of all requests for this object adapter.

See Also

`activate`, `hold`, `waitForDeactivate`, `Communicator::shutdown`.

find

```
Object find(Identity id);
```

Look up a servant in this object adapter's Active Servant Map by the identity of the Ice object it implements.

This operation only tries to lookup a servant in the Active Servant Map. It does not attempt to find a servant by using any installed `ServantLocator`.

Parameters

`id`

The identity of the Ice object for which the servant should be returned.

Return Value

The servant that implements the Ice object with the given identity, or null if no such servant has been found.

See Also

`Identity`, `findFacet`, `findByProxy`.

findAllFacets

```
FacetMap findAllFacets(Identity id);
```

Find all facets with the given identity in the Active Servant Map.

Parameters

id

The identity of the Ice object for which the facets should be returned.

Return Value

A collection containing all the facet names and servants which have been found, or an empty map if there is no facet for the given identity.

See Also

`find, findFacet.`

findByProxy

```
Object findByProxy(Object* proxy);
```

Look up a servant in this object adapter's Active Servant Map, given a proxy. This operation only tries to lookup a servant in the Active Servant Map. It does not attempt to find a servant via any installed `ServantLocators`.

Parameters

proxy

The proxy for which the servant should be returned.

Return Value

The servant that matches the proxy, or null if no such servant has been found.

See Also

`find, findFacet.`

findFacet

```
Object findFacet(Identity id, string facet);
```

Like `find`, but with a facet. Calling `find(id)` is equivalent to calling `findFacet` with an empty facet.

Parameters

`id`

The identity of the Ice object for which the servant should be returned.

`facet`

The facet. An empty facet means the default facet.

Return Value

The servant that implements the Ice object with the given identity and facet, or null if no such servant has been found.

See Also

`Identity`, `find`, `findByProxy`.

`findServantLocator`

```
ServantLocator findServantLocator(string category);
```

Find a Servant Locator installed with this object adapter.

Parameters

`category`

The category for which the Servant Locator can locate servants, or an empty string if the Servant Locator does not belong to any specific category.

Return Value

The Servant Locator, or null if no Servant Locator was found for the given category.

See Also

`Identity`, `addServantLocator`, `ServantLocator`.

`getCommunicator`

```
Communicator getCommunicator();
```

Get the communicator this object adapter belongs to.

Return Value

This object adapter's communicator.

See Also

`Communicator`.

`getName`

```
string getName();
```

Get the name of this object adapter.

Return Value

This object adapter's name.

`hold`

```
void hold();
```

Temporarily hold receiving and dispatching requests. The object adapter can be reactivated with the `activate` operation.

Holding is not immediate, i.e., after `hold` returns, the object adapter might still be active for some time. You can use `waitForHold` to wait until holding is complete.

See Also

`activate`, `deactivate`, `waitForHold`.

`remove`

```
Object remove(Identity id);
```

Remove a servant (that is, the default facet) from the object adapter's Active Servant Map.

Parameters

id

The identity of the Ice object that is implemented by the servant. If the servant implements multiple Ice objects, `remove` has to be called for all those Ice objects. Removing an identity that is not in the map throws `NotRegisteredException`.

Return Value

The removed servant.

See Also

`Identity`, `add`, `addWithUUID`.

removeAllFacets

```
FacetMap removeAllFacets(Identity id);
```

Remove all facets with the given identity from the Active Servant Map (that is, completely remove the Ice object, including its default facet). Removing an identity that is not in the map throws `NotRegisteredException`.

Parameters

id

The identity of the Ice object to be removed.

Return Value

A collection containing all the facet names and servants of the removed Ice object.

See Also

`remove`, `removeFacet`.

removeFacet

```
Object removeFacet(Identity id, string facet);
```

Like `remove`, but with a facet. Calling `remove(id)` is equivalent to calling `removeFacet` with an empty facet.

Parameters**id**

The identity of the Ice object that is implemented by the servant.

facet

The facet. An empty facet means the default facet.

Return Value

The removed servant.

See Also

[Identity](#), [addFacet](#), [addFacetWithUUID](#).

setLocator

```
void setLocator(Locator* loc);
```

Set an Ice locator for this object adapter. By doing so, the object adapter will register itself with the locator registry when it is activated for the first time. Furthermore, the proxies created by this object adapter will contain the adapter name instead of its endpoints.

Parameters**loc**

The locator used by this object adapter.

See Also

[createDirectProxy](#), [Locator](#), [LocatorRegistry](#).

waitForDeactivate

```
void waitForDeactivate();
```

Wait until the object adapter has deactivated. Calling `deactivate` initiates object adapter deactivation, and `waitForDeactivate` only returns when deactivation has been completed.

See Also`deactivate,waitForHold,Communicator::waitForShutdown.`**waitForHold**`void waitForHold();`

Wait until the object adapter holds requests. Calling `hold` initiates holding of requests, and `waitForHold` only returns when holding of requests has been completed.

See Also`hold,waitForDeactivate,Communicator::waitForShutdown.`

B.41 [Ice::ObjectAdapterDeactivatedException](#)**Overview**`local exception ObjectAdapterDeactivatedException`

This exception is raised if an attempt is made to use a deactivated `ObjectAdapter`.

See Also`ObjectAdapter::deactivate,Communicator::shutdown.`**name**`string name;`

Name of the adapter.

B.42 [Ice::ObjectAdapterIdInUseException](#)**Overview**`local exception ObjectAdapterIdInUseException`

This exception is raised if an `ObjectAdapter` cannot be activated because the `Locator` detected another active `ObjectAdapter` with the same adapter id.

`id`

`string id;`

Adapter id.

B.43 `Ice::ObjectFactory`

Overview

`local interface ObjectFactory`

A factory for objects. Object factories are used in several places, for example, when receiving "objects by value" and when `::Freeze` restores a persistent object. Object factories must be implemented by the application writer, and registered with the communicator.

Used By

`Communicator::addObjectFactory`, `Communicator::findObjectFactory`.

See Also

`::Freeze`.

`create`

`Object create(string type);`

Create a new object for a given object type. The type is the absolute type name, i.e., the name relative to the unnamed top-level Slice module. For example, the absolute type name for interfaces of type `Bar` in the module `Foo` is `::Foo::Bar`.

The leading `::` is required.

Parameters**type**

The object type.

Return Value

The object created for the given type, or nil if the factory is unable to create the object.

destroy

```
void destroy();
```

Called when the factory is removed from the communicator, or if the communicator is destroyed.

See Also

`Communicator::removeObjectFactory`, `Communicator::destroy`.

B.44 [Ice::ObjectNotExistException](#)**Overview**

```
local exception ObjectNotExistException extends  
RequestFailedException
```

This exception is raised if an object does not exist on the server, that is, if no facets with the given identity exist.

B.45 [Ice::ObjectNotFoundException](#)**Overview**

```
exception ObjectNotFoundException
```

This exception is raised if an object cannot be found.

B.46 `Ice::OperationMode`

Overview

`enum OperationMode`

The `OperationMode` determines the skeleton signature (for C++), as well as the retry behavior of the Ice run time for an operation invocation in case of a (potentially) recoverable error.

Used By

`Current::mode`.

Normal

`Normal`

Ordinary operations have `Normal` mode. These operations modify object state; invoking such an operation twice in a row has different semantics than invoking it once. The Ice run time guarantees that it will not violate at-most-once semantics for `Normal` operations.

Nonmutating

`Nonmutating`

Operations that use the Slice `Nonmutating` keyword must not modify object state. For C++, nonmutating operations generate `const` member functions in the skeleton. In addition, the Ice run time will attempt to transparently recover from certain run-time errors by re-issuing a failed request and propagate the failure to the application only if the second attempt fails.

Idempotent

`Idempotent`

Operations that use the Slice `Idempotent` keyword can modify object state, but invoking an operation twice in a row must result in the same object state as invoking it once. For example, `x = 1` is an idempotent statement, whereas `x += 1`

is not. For idempotent operations, the Ice run-time uses the same retry behavior as for nonmutating operations in case of a potentially recoverable error.

B.47 `Ice::OperationNotExistException`

Overview

```
local exception OperationNotExistException extends  
RequestFailedException
```

This exception is raised if an operation for a given object does not exist on the server. Typically this is caused by either the client or the server using an outdated Slice specification.

B.48 `Ice::Plugin`

Overview

```
local interface Plugin
```

A communicator plug-in. A plug-in generally adds a feature to a communicator, such as support for a protocol.

Derived Classes and Interfaces

```
::IceSSL::Plugin.
```

Used By

```
PluginManager::addPlugin, PluginManager::getPlugin.
```

destroy

```
void destroy();
```

Called when the communicator is being destroyed.

B.49 `Ice::PluginInitializationException`

Overview

```
local exception PluginInitializationException
```

This exception indicates that a failure occurred while initializing a plug-in.

reason

```
string reason;
```

The reason for the failure.

B.50 `Ice::PluginManager`

Overview

```
local interface PluginManager
```

Each communicator has a plugin manager to administer the set of plug-ins.

Used By

`Communicator::getPluginManager`.

addPlugin

```
void addPlugin(string name, Plugin pi);
```

Install a new plug-in.

Parameters

name

The plug-in's name.

pi

The plug-in.

destroy

```
void destroy();
```

Called when the communicator is being destroyed.

getPlugin

```
Plugin getPlugin(string name);
```

Obtain a plug-in by name.

Parameters

`name`

The plug-in's name.

Return Value

The plug-in.

B.51 Ice::Process

Overview

`interface Process`

An administrative interface for process management. Managed servers must implement this interface and invoke `ObjectAdapter::setProcess` to register the process proxy.

A servant implementing this interface is a potential target for denial-of-service attacks, therefore proper security precautions should be taken. For example, the servant can use a UUID to make its identity harder to guess, and be registered in an object adapter with a secured endpoint.

shutdown

```
void shutdown();
```

Initiate a graceful shutdown.

See Also

`Communicator::shutdown`.

B.52 `Ice::Properties`

Overview

`local interface Properties`

A property set used to configure Ice and Ice applications. Properties are key/value pairs, with both keys and values being `strings`. By convention, property keys should have the form *application-name*[.*category*[.*sub-category*]].*name*.

Used By

`Communicator::getProperties, clone`.

`clone`

`Properties clone();`

Create a copy of this property set.

Return Value

A copy of this property set.

`getCommandLineOptions`

`StringSeq getCommandLineOptions();`

Get a sequence of command-line options that is equivalent to this property set. Each element of the returned sequence is a command-line option of the form `--key=value`.

Return Value

The command line options for this property set.

getPropertiesForPrefix

```
PropertyDict getPropertiesForPrefix(string prefix);
```

Get all properties whose keys begins with *prefix*. If *prefix* is an empty string, then all properties are returned.

Return Value

The matching property set.

getProperty

```
string getProperty(string key);
```

Get a property by key. If the property does not exist, an empty string is returned.

Parameters

key

The property key.

Return Value

The property value.

See Also

`setProperty`.

getPropertyAsInt

```
int getPropertyAsInt(string key);
```

Get a property as an integer. If the property does not exist, 0 is returned.

Parameters

key

The property key.

Return Value

The property value interpreted as an integer.

See Also

`setProperty`.

getPropertyAsIntWithDefault

```
int getPropertyAsIntWithDefault(string key, int value);
```

Get a property as an integer. If the property does not exist, the given default value is returned.

Parameters

`key`

The property key.

`value`

The default value to use if the property does not exist.

Return Value

The property value interpreted as an integer, or the default value.

See Also

`setProperty`.

getPropertyWithDefault

```
string getPropertyWithDefault(string key, string value);
```

Get a property by key. If the property does not exist, the given default value is returned.

Parameters

`key`

The property key.

`value`

The default value to use if the property does not exist.

Return Value

The property value or the default value.

See Also

`setProperty`.

load

```
void load(string file);
```

Load properties from a file.

Parameters**file**

The property file.

parseCommandLineOptions

```
StringSeq parseCommandLineOptions(string prefix, StringSeq options);
```

Convert a sequence of command-line options into properties. All options that begin with `--prefix`. are converted into properties. If the prefix is empty, all options that begin with `--` are converted to properties.

Parameters**prefix**

The property prefix, or an empty string to convert all options starting with `--`.

options

The command-line options.

Return Value

The command-line options that do not start with the specified prefix, in their original order.

parseIceCommandLineOptions

```
StringSeq parseIceCommandLineOptions(StringSeq options);
```

Convert a sequence of command-line options into properties. All options that begin with one of the following prefixes are converted into properties: `--Ice`, `--`

`IceBox`, `--IcePack`, `--IcePatch`, `--IceSSL`, `--IceStorm`, `--Freeze`, and `--Glacier`.

Parameters

`options`

The command-line options.

Return Value

The command-line options that do not start with one of the listed prefixes, in their original order.

`setProperty`

```
void setProperty(string key, string value);
```

Set a property. To unset a property, set it to the empty string.

Parameters

`key`

The property key.

`value`

The property value.

See Also

`getProperty`.

B.53 `Ice::ProtocolException`

Overview

```
local exception ProtocolException
```

A generic exception base for all kinds of protocol error conditions.

Derived Exceptions

AbortBatchRequestException, BadMagicException, CloseConnectionException, CompressionException, CompressionNotSupportedException, ConnectionNotValidatedException, DatagramLimitException, IllegalMessageSizeException, MarshalException, UnknownMessageException, UnknownReplyStatusException, UnknownRequestIdException, UnsupportedEncodingException, UnsupportedProtocolException.

B.54 Ice::ProxyParseException

Overview

local exception ProxyParseException

This exception is raised if there was an error while parsing a stringified proxy.

str

string str;

The string that could not be parsed.

B.55 Ice::ProxyUnmarshalException

Overview

local exception ProxyUnmarshalException extends MarshalException

This exception is a specialization of MarshalException that is raised if inconsistent data is received while unmarshaling a proxy.

B.56 Ice::RequestFailedException

Overview

```
local exception RequestFailedException
```

This exception is raised if a request failed. This exception, and all exceptions derived from RequestFailedException, are transmitted by the Ice protocol, even though they are declared local.

Derived Exceptions

```
FacetNotExistException, ObjectNotExistException,  
OperationNotExistException.
```

facet

```
string facet;
```

The facet to which the request was sent.

id

```
Identity id;
```

The identity of the Ice Object to which the request was sent.

operation

```
string operation;
```

The operation name of the request.

B.57 Ice::Router

Overview

```
interface Router
```

The Ice router interface. Routers can be set either globally with `Communicator::setDefaultRouter`, or with `ice_router` on specific proxies.

The router interface is intended to be used by Ice internals and by router implementations. Regular user code should not attempt to use any functionality of this interface directly.

Derived Classes and Interfaces

`::Glacier::Router`.

`addProxy`

`void addProxy(Object* proxy);`

Add new proxy information to the router's routing table.

Parameters

`proxy`

The proxy to add.

`getClientProxy`

`Object* getClientProxy();`

Get the router's client proxy, i.e., the proxy to use for forwarding requests from the client to the router.

Return Value

The router's client proxy.

`getServerProxy`

`Object* getServerProxy();`

Get the router's server proxy, i.e., the proxy to use for forwarding requests from the server to the router.

Return Value

The router's server proxy.

B.58 `Ice::ServantLocator`

Overview

`local interface ServantLocator`

The servant locator, which is called by the object adapter to locate a servant that is not found in its active servant map.

Derived Classes and Interfaces

`::Freeze::Evictor.`

Used By

`ObjectAdapter::addServantLocator, ObjectAdapter::findServantLocator.`

See Also

`ObjectAdapter, ObjectAdapter::addServantLocator,`
`ObjectAdapter::findServantLocator.`

`deactivate`

`void deactivate(string category);`

Called when the object adapter in which this servant locator is installed is deactivated.

Parameters

`category`

Indicates for which category the servant locator is being deactivated.

See Also

`ObjectAdapter::deactivate, Communicator::shutdown, Communicator::destroy.`

`finished`

`void finished(Current curr, Object servant, LocalObject cookie);`

Called by the object adapter after a request has been made. This operation is only called if `locate` was called prior to the request and returned a non-null servant. This operation can be used for cleanup purposes after a request.

Parameters

`curr`

Information about the current operation call for which a servant was located by `locate`.

`servant`

The servant that was returned by `locate`.

`cookie`

The cookie that was returned by `locate`.

See Also

`ObjectAdapter`, `Current`, `locate`.

`locate`

```
Object locate(Current curr, out LocalObject cookie);
```

Called by the object adapter before a request is made when a servant cannot be found in the object adapter's active servant map. Note that the object adapter does not automatically insert the returned servant into its active servant map. This must be done by the servant locator implementation, if this is desired.

Important: If you call `locate` from your own code, you must also call `finished` when you have finished using the servant, provided that a non-null servant was returned. Otherwise you will get undefined behavior if you use Servant Locators such as the `::Freeze::Evictor`.

Parameters

`curr`

Information about the current operation for which a servant is required.

`cookie`

A "cookie" that will be passed to `finished`.

Return Value

The located servant, or null if no suitable servant has been found.

See Also

`ObjectAdapter.Current.finished`.

B.59 `Ice::ServerNotFoundException`**Overview**

`exception ServerNotFoundException`

This exception is raised if a server cannot be found.

B.60 `Ice::SocketException`**Overview**

`local exception SocketException extends SyscallException`

This exception is a specialization of `SyscallException` for socket errors.

Derived Exceptions

`ConnectFailedException, ConnectionLostException.`

B.61 `Ice::Stats`**Overview**

`local interface Stats`

An interface `Ice` uses to report statistics, such as how much data is sent or received. Applications must provide their own `Stats` by implementing this interface and installing it in a communicator.

Used By

`Communicator::getStats, Communicator::setStats.`

bytesReceived

`void bytesReceived(string protocol, int num);`

Callback to report that data has been received.

Parameters

`protocol`

The protocol over which data has been received (for example "tcp", "udp", or "ssl").

`num`

How many bytes have been received.

bytesSent

`void bytesSent(string protocol, int num);`

Callback to report that data has been sent.

Parameters

`protocol`

The protocol over which data has been sent (for example "tcp", "udp", or "ssl").

`num`

How many bytes have been sent.

B.62 `Ice::SyscallException`

Overview

local exception `SyscallException`

This exception is raised if a system error occurred in the server or client process. There are many possible causes for such a system exception. For details on the cause, `error` should be inspected.

Derived Exceptions

`SocketException`.

```
error  
int error;
```

The error number describing the system exception. For C++ and Unix, this is equivalent to `errno`. For C++ and Windows, this is the value returned by `GetLastError()` or `WSAGetLastError()`.

B.63 `Ice::TimeoutException`

Overview

```
local exception TimeoutException
```

This exception indicates a timeout condition.

Derived Exceptions

`CloseTimeoutException`, `ConnectTimeoutException`, `ConnectionTimeoutException`.

B.64 `Ice::TwowayOnlyException`

Overview

```
local exception TwowayOnlyException
```

This exception is raised if an attempt is made to invoke an operation with `ice_oneway`, `ice_batchOneway`, `ice_datagram`, or `ice_batchDatagram` and the operation has a return value, out parameters, or an exception specification.

```
operation
string operation;
```

The name of the operation that was invoked.

B.65 Ice::UnknownException

Overview

```
local exception UnknownException
```

This exception is raised if an operation call on a server raises an unknown exception. For example, for C++, this exception is raised if the server throws a C++ exception that is not directly or indirectly derived from `Ice::LocalException` or `Ice::UserException`.

Derived Exceptions

`UnknownLocalException`, `UnknownUserException`.

unknown

```
string unknown;
```

A textual representation of the unknown exception. This field may or may not be set, depending on the security policy of the server. Some servers may give this information to clients for debugging purposes, while others may not wish to disclose information about server internals.

B.66 Ice::UnknownLocalException

Overview

```
local exception UnknownLocalException extends UnknownException
```

This exception is raised if an operation call on a server raises a local exception. Because local exceptions are not transmitted by the Ice protocol, the client

receives all local exceptions raised by the server as `UnknownLocalException`. The only exception to this rule are all exceptions derived from `RequestFailedException`, which are transmitted by the Ice protocol even though they are declared `local`.

B.67 `Ice::UnknownMessageException`

Overview

`local exception UnknownMessageException extends ProtocolException`

This exception is a specialization of `ProtocolException`, indicating that an unknown protocol message has been received.

B.68 `Ice::UnknownReplyStatusException`

Overview

`local exception UnknownReplyStatusException extends ProtocolException`

This exception is a specialization of `ProtocolException`, indicating that an unknown reply status has been received.

B.69 `Ice::UnknownRequestIdException`

Overview

`local exception UnknownRequestIdException extends ProtocolException`

This exception is a specialization of `ProtocolException`, indicating that a response for an unknown request id has been received.

B.70 `Ice::UnknownUserException`

Overview

```
local exception UnknownUserException extends UnknownException
```

This exception is raised if an operation call on a server raises a user exception that is not declared in the exception's throws clause. Such undeclared exceptions are not transmitted from the server to the client by the Ice protocol, but instead the client just gets an `UnknownUserException`. This is necessary in order to not violate the contract established by an operation's signature: Only local exceptions and user exceptions declared in the throws clause can be raised.

B.71 `Ice::UnmarshalOutOfBoundsException`

Overview

```
local exception UnmarshalOutOfBoundsException extends  
MarshalException
```

This exception is a specialization of `MarshalException` that is raised if an out-of-bounds condition occurs during unmarshaling.

B.72 `Ice::UnsupportedEncodingException`

Overview

```
local exception UnsupportedEncodingException extends  
ProtocolException
```

This exception is a specialization of `ProtocolException`, indicating that an unsupported data encoding version has been encountered.

```
badMajor
```

```
int badMajor;
```

The major version number of the unsupported encoding.

badMinor

```
int badMinor;
```

The minor version number of the unsupported encoding.

major

```
int major;
```

The major version number of the encoding that is supported.

minor

```
int minor;
```

The highest minor version number of the encoding that can be supported.

B.73 `Ice::UnsupportedProtocolException`

Overview

```
local exception UnsupportedProtocolException extends  
ProtocolException
```

This exception is a specialization of `ProtocolException`, indicating that an unsupported protocol version has been encountered.

badMajor

```
int badMajor;
```

The major version number of the unsupported protocol.

badMinor

```
int badMinor;
```

The minor version number of the unsupported protocol.

major

```
int major;
```

The major version number of the protocol that is supported.

minor

```
int minor;
```

The highest minor version number of the protocol that can be supported.

B.74 `Ice::VersionMismatchException`

Overview

```
local exception VersionMismatchException
```

This exception is raised if the Ice library version does not match the Ice header files version.

B.75 `Freeze`

Overview

```
module Freeze
```

Freeze provides automatic persistence for Ice servants. Freeze provides a binary data format for maximum speed, as well as an XML data format for maximum flexibility. Freeze databases using the XML data format can be migrated when the description of the persistent data changes.

Key

```
sequence<byte> Key;
```

A database key, represented as a sequence of bytes.

Value

```
sequence<byte> Value;
```

A database value, represented as a sequence of bytes.

B.76 Freeze::Connection

Overview

```
local interface Connection
```

A connection to a database (database environment with Berkeley DB). If you want to use a connection concurrently in multiple threads, you need to serialize access to this connection.

beginTransaction

```
Transaction beginTransaction();
```

Create a new transaction. Only one transaction at a time can be associated with a connection.

Return Value

The new transaction

Exceptions

```
raises
```

TransactionAlreadyInProgressException if a transaction is already associated with this connection.

close

```
void close();
```

Closes this connection. If there is an associated transaction, it is rolled back.

currentTransaction

```
Transaction currentTransaction();
```

Returns the transaction associated with this connection.

Return Value

The current transaction if there is one, null otherwise.

getCommunicator

```
::Ice::Communicator getCommunicator();
```

Returns the communicator associated with this connection

getName

```
string getName();
```

The name of the connected system (e.g. Berkeley DB environment)

B.77 Freeze::DatabaseException

Overview

```
local exception DatabaseException
```

A Freeze database exception.

Derived Exceptions

DeadlockException, NotFoundException.

See Also

DB, Evictor, Connection.

message

```
string message;
```

A message describing the reason for the exception.

B.78 `Freeze::DeadlockException`

Overview

```
local exception DeadlockException extends DatabaseException
```

A Freeze database deadlock exception. Applications can react to this exception by aborting and trying the transaction again.

B.79 `Freeze::Evictor`

Overview

```
local interface Evictor extends ::Ice::ServantLocator
```

An automatic Ice object persistence manager, based on the evictor pattern. The evictor is a servant locator implementation that stores the persistent state of its objects in a database. Any number of objects can be registered with an evictor, but only a configurable number of servants are active at a time. These active servants reside in a queue; the least recently used servant in the queue is the first to be evicted when a new servant is activated.

See Also

`ServantInitializer`.

`add`

```
Object* add(Object servant, ::Ice::Identity id);
```

Add a servant to this evictor. The state of the servant passed to this operation will be saved in the evictor's persistent store.

Parameters

`servant`

The servant to add.

`id`

The identity of the Ice object that is implemented by the servant.

Return Value

A proxy that matches the given identity and this evictor's object adapter.

Exceptions

`AlreadyRegisteredException`

Raised if the evictor already has an object with this identity.

`DatabaseException`

Raised if a database failure occurred.

`EvictorDeactivatedException`

Raised if the evictor has been deactivated.

See Also

`addFacet`, `remove`, `removeFacet`.

`addFacet`

```
Object* addFacet(Object servant, ::Ice::Identity id, string facet);
```

Like `add`, but with a facet. Calling `add(servant, id)` is equivalent to calling `addFacet` with an empty facet.

Parameters

`servant`

The servant to add.

`id`

The identity of the Ice object that is implemented by the servant.

`facet`

The facet. An empty facet means the default facet.

Return Value

A proxy that matches the given identity and this evictor's object adapter.

Exceptions

`AlreadyRegisteredException`

Raised if the evictor already has an object with this identity.

`DatabaseException`

Raised if a database failure occurred.

`EvictorDeactivatedException`

Raised if the evictor has been deactivated.

See Also

`add`, `remove`, `removeFacet`.

`createObject`

```
void createObject(::Ice::Identity id, Object servant);
```

Add or update a servant. The state of the servant passed to this operation will be saved in the evictor's persistent store. This operation is deprecated and will be removed in a future release. It is recommended to use `add` instead.

Parameters

`id`

The identity of the Ice object that is implemented by the servant.

`servant`

The servant to add.

Exceptions

`DatabaseException`

Raised if a database failure occurred.

`EvictorDeactivatedException`

Raised if the evictor has been deactivated.

See Also

`add, destroyObject.`

destroyObject

```
void destroyObject(::Ice::Identity id);
```

Permanently destroy an Ice object. Like `remove`, except `destroyObject` does not raise any exception when the object does not exist. This operation is deprecated and will be removed in a future release. It is recommended to use `remove` instead.

Parameters

`id`

The identity of the Ice object.

Exceptions

`DatabaseException`

Raised if a database failure occurred.

`EvictorDeactivatedException`

Raised if the evictor has been deactivated.

See Also

`remove, createObject.`

getIterator

```
EvictorIterator getIterator(string facet, int batchSize);
```

Get an iterator for the identities managed by the evictor.

Parameters

`facet`

The facet. An empty facet means the default facet.

`batchSize`

Internally, the Iterator retrieves the identities in batches of size `batchSize`.

Selecting a small `batchSize` can have an adverse effect on performance.

Return Value

A new iterator.

Exceptions

`EvictorDeactivatedException`

Raised if a the evictor has been deactivated.

getSize

```
int getSize();
```

Get the size of the evictor's servant queue.

Return Value

The size of the servant queue.

Exceptions

`EvictorDeactivatedException`

Raised if a the evictor has been deactivated.

See Also

`setSize`.

hasFacet

```
bool hasFacet(::Ice::Identity id, string facet);
```

Like `hasObject`, but with a facet. Calling `hasObject(id)` is equivalent to calling `hasFacet` with an empty facet.

Return Value

true if the identity is managed by the evictor for the given facet, false otherwise.

Exceptions

`DatabaseException`

Raised if a database failure occurred.

EvictorDeactivatedException

Raised if a the evictor has been deactivated.

hasObject

```
bool hasObject(::Ice::Identity id);
```

Returns true if the given identity is managed by the evictor with the default facet.

Return Value

true if the identity is managed by the evictor, false otherwise.

Exceptions**DatabaseException**

Raised if a database failure occurred.

EvictorDeactivatedException

Raised if a the evictor has been deactivated.

keep

```
void keep(::Ice::Identity id);
```

Lock this object in the evictor cache. This lock can be released by `release` or `remove`. `release` releases only one lock, while `remove` releases all the locks.

Parameters**id**

The identity of the Ice object.

Exceptions**NotRegisteredException**

Raised if this identity was not registered with the evictor.

DatabaseException

Raised if a database failure occurred.

See Also

`keepFacet`, `release`, `remove`.

keepFacet

```
void keepFacet(::Ice::Identity id, string facet);
```

Like `keep`, but with a facet. Calling `keep(id)` is equivalent to calling `keepFacet` with an empty facet.

Parameters**id**

The identity of the Ice object.

facet

The facet. An empty facet means the default facet.

Exceptions**NotRegisteredException**

Raised if this identity was not registered with the evictor.

DatabaseException

Raised if a database failure occurred.

See Also

`keep`, `releaseFacet`, `removeFacet`.

release

```
void release(::Ice::Identity id);
```

Release a "lock" acquired by `keep`. Once all the locks on an object have been released, the object is again subject to the normal eviction strategy.

Parameters**id**

The identity of the Ice object.

Exceptions**NotRegisteredException**

Raised if this object was not "locked" with `keep` or `keepFacet`.

See Also

`keepFacet`, `release`.

releaseFacet

```
void releaseFacet(::Ice::Identity id, string facet);
```

Like `release`, but with a facet. Calling `release(id)` is equivalent to calling `releaseFacet` with an empty facet.

Parameters

`id`

The identity of the Ice object.

`facet`

The facet. An empty facet means the default facet.

Exceptions

`NotRegisteredException`

Raised if this object was not "locked" with `keep` or `keepFacet`.

See Also

`keep`, `releaseFacet`.

remove

```
Object remove(::Ice::Identity id);
```

Permanently destroy an Ice object.

Parameters

`id`

The identity of the Ice object.

Return Value

The removed servant.

Exceptions

NotRegisteredException

Raised if this identity was not registered with the evictor.

DatabaseException

Raised if a database failure occurred.

EvictorDeactivatedException

Raised if the evictor has been deactivated.

See Also

`add`, `removeFacet`.

`removeFacet`

`Object removeFacet(::Ice::Identity id, string facet);`

Like `remove`, but with a facet. Calling `remove(id)` is equivalent to calling `removeFacet` with an empty facet.

Parameters

`id`

The identity of the Ice object.

`facet`

The facet. An empty facet means the default facet.

Return Value

The removed servant.

Exceptions

NotRegisteredException

Raised if this identity was not registered with the evictor.

DatabaseException

Raised if a database failure occurred.

EvictorDeactivatedException

Raised if the evictor has been deactivated.

See Also

`remove`, `addFacet`.

setSize

```
void setSize(int sz);
```

Set the size of the evictor's servant queue. This is the maximum number of servants the evictor keeps active. Requests to set the queue size to a value smaller than zero are ignored.

Parameters

`sz`

The size of the servant queue. If the evictor currently holds more than `setSize` servants in its queue, it evicts enough servants to match the new size. Note that this operation can block if the new queue size is smaller than the current number of servants that are servicing requests. In this case, the operation waits until a sufficient number of servants complete their requests.

Exceptions

`EvictorDeactivatedException`

Raised if a the evictor has been deactivated.

See Also

`getSize`.

B.80 `Freeze::EvictorDeactivatedException`

Overview

local exception `EvictorDeactivatedException`

This exception is raised if the evictor has been deactivated.

B.81 `Freeze::EvictorIterator`

Overview

```
local interface EvictorIterator
```

An iterator for the objects managed by the evictor. Note that an `EvictorIterator` is not thread-safe: the application needs to serialize access to a given `EvictorIterator`, for example by using it in just one thread.

Used By

`Evictor::getIterator`.

See Also

`Evictor`.

`hasNext`

```
bool hasNext();
```

Determines if the iteration has more elements.

Return Value

True if the iterator has more elements, false otherwise.

Exceptions

`DatabaseException`

Raised if a database failure occurs while retrieving a batch of objects.

`next`

```
::Ice::Identity next();
```

Obtains the next identity in the iteration.

Return Value

s The next identity in the iteration.

Exceptions

`NoSuchElementException`

Raised if there is no further elements in the iteration.

`DatabaseException`

Raised if a database failure occurs while retrieving a batch of objects.

B.82 `Freeze::InvalidPositionException`

Overview

`local exception InvalidPositionException`

This Freeze Iterator is not on a valid position, for example this position has been erased.

B.83 `Freeze::NoSuchElementException`

Overview

`local exception NoSuchElementException`

This exception is raised if there are no further elements in the iteration.

B.84 `Freeze::NotFoundException`

Overview

`local exception NotFoundException extends DatabaseException`

A Freeze database exception, indicating that a database record could not be found.

B.85 `Freeze::ObjectRecord`

Overview

```
struct ObjectRecord
```

The evictor uses a number of maps `::Ice::Identity` to `ObjectRecord` as its persistent storage.

```
servant
```

```
Object servant;
```

```
stats
```

```
Statistics stats;
```

B.86 `Freeze::ServantInitializer`

Overview

```
local interface ServantInitializer
```

A servant initializer is installed in an evictor and provides the application with an opportunity to perform custom servant initialization.

See Also

`Evictor`.

```
initialize
```

```
void initialize(::Ice::ObjectAdapter adapter, ::Ice::Identity  
identity, string facet, Object servant);
```

Called whenever the evictor creates a new servant. This operation allows application code to perform custom servant initialization after the servant has been created by the evictor and its persistent state has been restored.

Parameters

`adapter`

The object adapter in which the evictor is installed.

`identity`

The identity of the Ice object for which the servant was created.

`facet`

The facet. An empty facet means the default facet.

`servant`

The servant to initialize.

See Also

`::Ice::Identity`.

B.87 `Freeze::Statistics`

Overview

`struct Statistics`

The evictor maintains statistics about each object.

Used By

`ObjectRecord::stats`.

`avgSaveTime`

`long avgSaveTime;`

The average time between saves, in milliseconds.

creationTime

```
long creationTime;
```

The time the object was created, in milliseconds since Jan 1, 1970 0:00.

lastSaveTime

```
long lastSaveTime;
```

The time the object was last saved, in milliseconds relative to `creationTime`.

B.88 `Freeze::Transaction`

Overview

local interface Transaction

A transaction. If you want to use a transaction concurrently in multiple threads, you need to serialize access to this transaction.

Used By

`Connection::beginTransaction`, `Connection::currentTransaction`.

commit

```
void commit();
```

Commit this transaction.

rollback

```
void rollback();
```

Roll back this transaction.

B.89 `Freeze::TransactionAlreadyInProgressException`

Overview

```
local exception TransactionAlreadyInProgressException
```

B.90 `IceBox`

Overview

```
module IceBox
```

`IceBox` is an application server specifically for Ice applications. `IceBox` can easily run and administer Ice services that are dynamically loaded as a DLL, shared library, or Java class.

B.91 `IceBox::FailureException`

Overview

```
local exception FailureException
```

Indicates a failure occurred. For example, if a service encounters an error during initialization, or if the service manager is unable to load a service executable.

`reason`

```
string reason;
```

The reason for the failure.

B.92 `IceBox::FreezeService`

Overview

Local interface `FreezeService` extends `ServiceBase`

A Freeze application service managed by a `ServiceManager`.

See Also

`ServiceBase`.

`start`

```
void start(string name, ::Ice::Communicator communicator,
           ::Ice::StringSeq args, string envName);
```

Start the service. The given communicator is created by the `ServiceManager` for use by the service. This communicator may also be used by other services, depending on the service configuration. The database environment is created by the `ServiceManager` for the exclusive use of the service.

The `ServiceManager` owns the communicator and the database environment, and is responsible for destroying them.

Parameters

`name`

The service's name, as determined by the configuration.

`communicator`

A communicator for use by the service.

`args`

The service arguments that were not converted into properties.

`envName`

The name of the Freeze database environment.

Exceptions

`FailureException`

Raised if `start` failed.

B.93 `IceBox::Service`

Overview

local interface Service extends ServiceBase

A standard application service managed by a ServiceManager.

See Also

`ServiceBase`.

`start`

```
void start(string name, ::Ice::Communicator communicator,
           ::Ice::StringSeq args);
```

Start the service. The given communicator is created by the ServiceManager for use by the service. This communicator may also be used by other services, depending on the service configuration.

The ServiceManager owns this communicator, and is responsible for destroying it.

Parameters

`name`

The service's name, as determined by the configuration.

`communicator`

A communicator for use by the service.

`args`

The service arguments that were not converted into properties.

Exceptions

`FailureException`

Raised if `start` failed.

B.94 IceBox::ServiceBase

Overview

local interface ServiceBase

Base interface for an application service managed by a ServiceManager.

Derived Classes and Interfaces

FreezeService, Service.

See Also

ServiceManager, Service, FreezeService.

stop

void stop();

Stop the service.

B.95 IceBox::ServiceManager

Overview

interface ServiceManager

Administers a set of Service instances.

See Also

Service.

shutdown

void shutdown();

Shutdown all services. This will cause Service::stop to be invoked on all configured services.

B.96 IcePack

Overview

```
module IcePack
```

IcePack is a server activation and deployment tool. IcePack, simplifies the complex task of deploying applications in a heterogenous computer network.

B.97 IcePack::AdapterDeploymentException

Overview

```
exception AdapterDeploymentException extends DeploymentException
```

This exception is raised if an error occurs during adapter registration.

```
id
```

```
string id;
```

The id of the adapter that could not be registered.

B.98 IcePack::AdapterNotExistException

Overview

```
exception AdapterNotExistException
```

This exception is raised if an adapter does not exist.

B.99 IcePack::Admin

Overview

`interface Admin`

The IcePack administrative interface.

Warning: *Allowing access to this interface is a security risk! Please see the IcePack documentation for further information.*

`addApplication`

```
void addApplication(string descriptor, ::Ice::StringSeq targets)  
throws DeploymentException;
```

Add an application to IcePack. An application is a set of servers.

Parameters

`descriptor`

The application descriptor.

`targets`

The optional targets to deploy. A target is a list of components separated by dots and a target name. For example, the "debug" target of "service1" in "server1" will be deployed if the target "server1.service1.debug" is specified.

Exceptions

`DeploymentException`

Raised if application deployment failed.

See Also

`removeApplication`.

`addObject`

```
void addObject(Object* obj) throws ObjectExistsException,  
ObjectDeploymentException;
```

Add an object to the object registry. IcePack will get the object type by calling `ice_id` on the given proxy. The object must be reachable.

Parameters

`obj`

The object to be added to the registry.

Exceptions

`ObjectDeploymentException`

Raised if an error occurred while trying to register this object. This can occur if the type of the object cannot be determined because the object is not reachable.

`ObjectExistsException`

Raised if the object is already registered.

`addObjectWithType`

```
void addObjectWithType(Object* obj, string type) throws  
ObjectExistsException;
```

Add an object to the object registry and explicitly specify its type.

Parameters

`obj`

The object to be added to the registry.

`type`

The object type.

Exceptions

`ObjectExistsException`

Raised if the object is already registered.

`addServer`

```
void addServer(string node, string name, string path, string  
libraryPath, string descriptor, ::Ice::StringSeq targets) throws  
DeploymentException, NodeUnreachableException;
```

Add a server to an IcePack node.

Parameters

node

The name of the node where the server will be deployed.

name

The server name.

path

The server path. For C++ servers, this is the path of the executable. For C++ icebox, this is the path of the C++ icebox executable or, if empty, IcePack will rely on the PATH to find it. For a Java server or Java icebox, this is the path of the java command or, if empty, IcePack will rely on the PATH to find it.

librarypath

Specify the LD_LIBRARY_PATH value for C++ servers or the CLASSPATH value for Java servers.

descriptor

The server deployment descriptor.

targets

The optional targets to deploy. A target is a list of components separated by dots and a target name. For example, the "debug" target of "service1" in "server1" will be deployed if the target "server1.service1.debug" is specified.

Exceptions

DeploymentException

Raised if server deployment failed.

NodeUnreachableException

Raised if the node could not be reached.

See Also

`removeServer`.

getAdapterEndpoints

```
string getAdapterEndpoints(string id) throws  
AdapterNotExistException, NodeUnreachableException;
```

Get the list of endpoints for an adapter.

Parameters

`id`

The adapter id.

Return Value

The stringified adapter endpoints.

Exceptions

`AdapterNotExistException`

Raised if the adapter is not found.

`getAllAdapterIds`

`::Ice::StringSeq getAllAdapterIds();`

Get all the adapter ids registered with IcePack.

Return Value

The adapter ids.

`getAllNodeNames`

`::Ice::StringSeq getAllNodeNames();`

Get all the IcePack nodes currently registered.

Return Value

The node names.

`getAllServerNames`

`::Ice::StringSeq getAllServerNames();`

Get all the server names registered with IcePack.

Return Value

The server names.

See Also

`getServerDescription`, `getServerState`.

getServerActivation

```
ServerActivation getServerActivation(string name) throws  
ServerNotExistException, NodeUnreachableException;
```

Get the server's activation mode.

Parameters

`name`

Must match the name of `ServerDescription::name`.

Return Value

The server activation mode.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`getServerDescription`, `getServerState`, `getAllServerNames`.

getServerDescription

```
ServerDescription getServerDescription(string name) throws  
ServerNotExistException, NodeUnreachableException;
```

Get a server's description.

Parameters**name**

Must match the name of `ServerDescription::name`.

Return Value

The server description.

Exceptions`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also`getServerState`, `getServerPid`, `getAllServerNames`.**getServerPid**

```
int getServerPid(string name) throws ServerNotExistException,  
NodeUnreachableException;
```

Get a server's system process id. The process id is operating system dependent.

Parameters**name**

Must match the name of `ServerDescription::name`.

Return Value

The server process id.

Exceptions`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`getServerDescription`, `getServerState`, `getAllServerNames`.

getServerState

```
ServerState getServerState(string name) throws  
ServerNotExistException, NodeUnreachableException;
```

Get a server's state.

Parameters

`name`

Must match the name of `ServerDescription::name`.

Return Value

The server state.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`getServerDescription`, `getServerPid`, `getAllServerNames`.

pingNode

```
bool pingNode(string name) throws NodeNotExistException;
```

Ping an IcePack node to see if it is active.

Return Value

true if the node ping succeeded, false otherwise.

removeApplication

```
void removeApplication(string descriptor) throws  
DeploymentException;
```

Remove an application from IcePack.

Parameters

descriptor

The application descriptor.

See Also

[addApplication](#).

removeObject

```
void removeObject(Object* obj) throws ObjectNotExistException;
```

Remove an object from the object registry.

Parameters

obj

The object to be removed from the registry.

Exceptions

[ObjectNotExistException](#)

Raised if the object cannot be found.

removeServer

```
void removeServer(string name) throws DeploymentException,  
ServerNotExistException, NodeUnreachableException;
```

Remove a server from an IcePack node.

Parameters

name

Must match the name of [ServerDescription::name](#).

Exceptions

`DeploymentException`

Raised if the server deployer failed to remove the server.

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`addServer`.

`sendSignal`

```
void sendSignal(string name, string signal) throws  
ServerNotExistException, NodeUnreachableException,  
BadSignalException;
```

Send signal to a server.

Parameters

`name`

Must match the name of `ServerDescription::name`.

`signal`

The signal, for example SIGTERM or 15.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

`BadSignalException`

Raised if the signal is not recognized by the target server.

setServerActivation

```
void setServerActivation(string name, ServerActivation mode)
throws ServerNotExistException, NodeUnreachableException;
```

Set the server's activation mode.

Parameters

name

Must match the name of `ServerDescription::name`.

Return Value

The server activation mode.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`getServerDescription`, `getServerState`, `getAllServerNames`.

shutdown

```
void shutdown();
```

Shut down the IcePack registry.

shutdownNode

```
void shutdownNode(string name) throws NodeNotExistException;
```

Shutdown an IcePack node.

startServer

```
bool startServer(string name) throws ServerNotExistException,
NodeUnreachableException;
```

Start a server.

Parameters

name

Must match the name of `ServerDescription::name`.

Return Value

True if the server was successfully started, false otherwise.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

`stopServer`

```
void stopServer(string name) throws ServerNotExistException,  
NodeUnreachableException;
```

Stop a server.

Parameters

name

Must match the name of `ServerDescription::name`.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

`writeMessage`

```
void writeMessage(string name, string message, int fd) throws  
ServerNotExistException, NodeUnreachableException;
```

Write message on server stdout or stderr

Parameters

`name`

Must match the name of `ServerDescription::name`.

`message`

The message.

`fd`

1 for stdout, 2 for stderr.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

B.100 `IcePack::BadSignalException`

Overview

`exception BadSignalException`

This exception is raised if an unknown signal was sent to a server.

B.101 `IcePack::DeploymentException`

Overview

`exception DeploymentException`

A generic exception base for all kinds of deployment error exception.

Derived Exceptions

`AdapterDeploymentException`, `ObjectDeploymentException`, `ParserDeploymentException`, `ServerDeploymentException`.

component

```
string component;
```

The path of the component that caused the deployment to fail. The path is a dot-separated list of component names. It always starts with the node name, followed by the server name, and finally the service name.

reason

```
string reason;
```

The reason for the failure.

B.102 `IcePack::NodeNotExistException`

Overview

```
exception NodeNotExistException
```

This exception is raised if a node does not exist.

B.103 `IcePack::NodeUnreachableException`

Overview

```
exception NodeUnreachableException
```

This exception is raised if a node could not be reached.

B.104 IcePack::ObjectDeploymentException

Overview

```
exception ObjectDeploymentException extends DeploymentException
```

This exception is raised if an error occurs during object registration.

proxy

```
Object* proxy;
```

The object that could not be registered.

B.105 IcePack::ObjectExistsException

Overview

```
exception ObjectExistsException
```

This exception is raised if an object already exists.

B.106 IcePack::ObjectNotExistException

Overview

```
exception ObjectNotExistException
```

This exception is raised if an object does not exist.

B.107 IcePack::ParserDeploymentException

Overview

```
exception ParserDeploymentException extends DeploymentException
```

This exception is raised if an error occurs while parsing the XML descriptor of a component.

B.108 IcePack::Query

Overview

`interface Query`

The IcePack query interface. This interface is accessible to Ice clients who wish to lookup objects.

`findAllObjectsWithType`

```
::Ice::ObjectProxySeq findAllObjectsWithType(string type) throws
ObjectNotExistException;
```

Find all the objects with the given type.

Parameters

`type`

The object type.

Return Value

The proxies.

Exceptions

`ObjectNotExistException`

Raised if no objects can be found.

`findObjectById`

```
Object* findObjectById(::Ice::Identity id) throws
ObjectNotExistException;
```

Find an object by identity.

Parameters`id`

The identity.

Return Value

The proxy.

Exceptions`ObjectNotExistException`

Raised if no objects can be found.

findObjectType

```
Object* findObjectType(string type) throws  
ObjectNotExistException;
```

Find an object by type.

Parameters`type`

The object type.

Return Value

The proxy.

Exceptions`ObjectNotExistException`

Raised if no objects can be found.

B.109 `IcePack::ServerActivation`

Overview`enum ServerActivation`

The server activation mode.

Used By

```
Admin::getServerActivation, Admin::setServerActivation.
```

OnDemand

OnDemand

The server is activated on demand if a client requests one of the server's adapter endpoints and the server is not already running.

Manual

Manual

The server is activated manually through the administrative interface.

B.110 [**IcePack::ServerDeploymentException**](#)**Overview**

```
exception ServerDeploymentException extends DeploymentException
```

This exception is raised if an error occurs while deploying a server.

server

```
string server;
```

The name of the server that could not be deployed.

B.111 [**IcePack::ServerDescription**](#)**Overview**

```
struct ServerDescription
```

The server description.

Used By

`Admin::getServerDescription.`

args

`::Ice::StringSeq args;`

The server arguments.

descriptor

`string descriptor;`

The path of the deployment descriptor used to deploy the server.

envs

`::Ice::StringSeq envs;`

The server environment variables.

name

`string name;`

Server name.

node

`string node;`

The name of the node where the server is deployed.

path

`string path;`

The server path.

pwd

```
string pwd;
```

The server working directory.

serviceManager

```
::IceBox::ServiceManager* serviceManager;
```

The IceBox service manager proxy if the server is an IceBox service, otherwise a null proxy.

targets

```
::Ice::StringSeq targets;
```

Targets used to deploy the server.

B.112 `IcePack::ServerNotExistException`

Overview

```
exception ServerNotExistException
```

This exception is raised if a server does not exist.

B.113 `IcePack::ServerState`

Overview

```
enum ServerState
```

An enumeration representing the state of the server.

Used By

`Admin::getServerState`.

Inactive

Inactive

The server is not running.

Activating

Activating

The server is being activated and will change to the active state if the server fork succeeded or to the Inactive state if it failed.

Active

Active

The server is running.

Deactivating

Deactivating

The server is being deactivated.

Destroying

Destroying

The server is being destroyed.

Destroyed

Destroyed

The server is destroyed.

B.114 IceSSL

Overview

```
module IceSSL
```

IceSSL is a dynamic SSL transport plug-in for the Ice core. It provides authentication, encryption, and message integrity, using the industry-standard SSL protocol.

B.115 IceSSL::CertificateException

Overview

```
local exception CertificateException extends SslException
```

A root exception class for all exceptions related to public key certificates.

Derived Exceptions

```
CertificateParseException, CertificateSignatureException, CertificateSigningException.
```

B.116 IceSSL::CertificateKeyMatchException

Overview

```
local exception CertificateKeyMatchException extends ContextException
```

When loading a public and private key pair into a `Context`, the load succeeded, but the private key and public key (certificate) did not match.

B.117 `IceSSL::CertificateLoadException`

Overview

```
local exception CertificateLoadException extends ContextException
```

Indicates that a problem occurred while loading a certificate into a Context from either a memory buffer or from a file.

B.118 `IceSSL::CertificateParseException`

Overview

```
local exception CertificateParseException extends  
CertificateException
```

Indicates that IceSSL was unable to parse the provided public key certificate into a form usable by the underlying SSL implementation.

B.119 `IceSSL::CertificateSignatureException`

Overview

```
local exception CertificateSignatureException extends  
CertificateException
```

Indicates that the signature verification of a newly signed temporary RSA certificate has failed.

B.120 `IceSSL::CertificateSigningException`

Overview

```
local exception CertificateSigningException extends  
CertificateException
```

Indicates that a problem occurred while signing certificates during temporary RSA certificate generation.

B.121 `IceSSL::CertificateVerificationException`

Overview

```
local exception CertificateVerificationException extends  
ShutdownException
```

Indicates a problem occurred during the certificate verification phase of the SSL handshake. This is currently only thrown by server connections.

B.122 `IceSSL::CertificateVerifier`

Overview

```
local interface CertificateVerifier
```

The `CertificateVerifier` is the base interface for all classes that define additional application-specific certificate verification rules. These rules are evaluated during the SSL handshake by an instance of a class derived from `CertificateVerifier`. The methods defined in derived interfaces will depend upon the requirements of the underlying SSL implementation. Default certificate verifier implementations can be obtained via the `Plugin`. As this is simply a base class for purposes of derivation, no methods are defined.

Used By

```
Plugin::getDefaultCertVerifier, Plugin::getSingleCertVerifier,  
Plugin::setCertificateVerifier.
```

See Also

`Plugin`.

setContext

```
void setContext(ContextType type);
```

Set the context type of this Certificate Verifier.

Parameters

type

The type of context that is using this CertificateVerifier, Client, Server or ClientServer.

B.123 **IceSSL::CertificateVerifierTypeException**

Overview

```
local exception CertificateVerifierTypeException extends SslException
```

This exception indicates that the provided CertificateVerifier was not derived from the proper base class, and thus, does not provide the appropriate interface.

B.124 **IceSSL::ConfigParseException**

Overview

```
local exception ConfigParseException extends SslException
```

This exception indicates that a problem occurred while parsing the SSL configuration file, or while attempting to locate the configuration file. This exception could indicate a problem with the `IceSSL.Client.Config`, `IceSSL.Server.Config`, `IceSSL.Client.CertPath` or `IceSSL.Server.CertPath` properties for your `::Ice::Communicator`.

B.125 IceSSL::ConfigurationLoadingException

Overview

```
local exception ConfigurationLoadingException extends SslException
```

This exception indicates that an attempt was made to load the configuration for a Context, but the property specifying the indicated Context's SSL configuration file was not set. Check the values for the appropriate property, either `IceSSL.Client.Config` or `IceSSL.Server.Config`.

B.126 IceSSL::ContextException

Overview

```
local exception ContextException extends SslException
```

A problem was encountered while setting up the Context. This can include problems related to loading certificates and keys or calling methods on a Context that has not been initialized as of yet.

Derived Exceptions

```
CertificateKeyMatchException, CertificateLoadException, ContextInitializationException, ContextNotConfiguredException, PrivateKeyLoadException, TrustedCertificateAddException, UnsupportedContextException.
```

B.127 IceSSL::ContextInitializationException

Overview

```
local exception ContextInitializationException extends ContextException
```

Indicates that a problem occurred while initializing the context structure of the underlying SSL implementation.

B.128 IceSSL::ContextNotConfiguredException

Overview

```
local exception ContextNotConfiguredException extends  
ContextException
```

This exception is raised when an attempt is made to make use of a Context that has not been configured yet.

B.129 IceSSL::ContextType

Overview

```
enum ContextType
```

A Plugin may serve as a Client, Server or both (ClientServer). A Context is set up inside the Plugin in order to handle either Client or Server roles. The Context represents a role-specific configuration. Some Plugin operations require a ContextType argument to identify the Context.

Used By

```
CertificateVerifier::setContext, Plugin::addTrustedCertificate,  
Plugin::addTrustedCertificateBase64, Plugin::configure,  
Plugin::loadConfig, Plugin::setCertificateVerifier, Plugin::setRSAKeys,  
Plugin::setRSAKeysBase64.
```

Client

Client

Select only the Client Context, no modifications to the Server.

Server

Server

Select only the Server Context, no modifications to the Client.

`ClientServer`

`ClientServer`

Select and affect changes on both the Client and Server Contexts.

B.130 `IceSSL::Plugin`

Overview

```
local interface Plugin extends ::Ice::Plugin
```

The interface for the SSL plug-in. This interface is typically used to perform programmatic configuration of the plug-in.

`addTrustedCertificate`

```
void addTrustedCertificate(ContextType cType, ::Ice::ByteSeq  
certificate);
```

Add a trusted certificate to the plug-in's default certificate store. The provided certificate (passed in binary DER format) is added to the trust list so that the certificate, and all certificates signed by its private key, are trusted. This method only affects new connections -- existing connections are left unchanged.

Parameters

`contextType`

The Context(s) in which to add the trusted certificate.

`certificate`

The certificate, in binary DER format, to be trusted.

`addTrustedCertificateBase64`

```
void addTrustedCertificateBase64(ContextType cType, string  
certificate);
```

Add a trusted certificate to the plug-in's default certificate store. The provided certificate (passed in Base64-encoded binary DER format, as per the PEM format)

is added to the trust list so that the certificate, and all certificates signed by its private key, are trusted. This method only affects new connections -- existing connections are left unchanged.

Parameters

`contextType`

The Context(s) in which to add the trusted certificate.

`certificate`

The certificate to be trusted, in Base64-encoded binary DER format.

`configure`

```
void configure(ContextType cType);
```

Configure the plug-in. If the plug-in is left in an unconfigured state, it will load its configuration from the properties `IceSSL.Server.Config` or `IceSSL.Client.Config`, depending on the context type. Configuration property settings will also be loaded during this operation, with the property values overriding those of the configuration file.

Parameters

`contextType`

The Context(s) to configure.

`getDefaultCertVerifier`

```
CertificateVerifier getDefaultCertVerifier();
```

Retrieves an instance of the `CertificateVerifier` that is installed by default in all plug-in instances.

Return Value

`CertificateVerifier`

`getSingleCertVerifier`

```
CertificateVerifier getSingleCertVerifier(::Ice::ByteSeq  
certificate);
```

Returns an instance of a `CertificateVerifier` that only accepts a single certificate, that being the RSA certificate represented by the binary DER encoding contained in the provided byte sequence. This is useful if you wish your application to accept connections from one party.

Be sure to use the peer `verifyMode` in your SSL configuration file.

Parameters

`certificate`

A DER encoded RSA certificate.

Return Value

`CertificateVerifier`

`loadConfig`

```
void loadConfig(ContextType cType, string configFile, string certPath);
```

Configure the plug-in for the given Context using the settings in the given configuration file. If the plug-in is left in an unconfigured state, it will load its configuration from the property `IceSSL.Server.Config` or `IceSSL.Client.Config`, depending on the context type. Configuration property settings will also be loaded as part of this operation, with the property values overriding those of the configuration file.

Parameters

`contextType`

The Context to configure.

`configFile`

The file containing the SSL configuration information.

`certPath`

The path where certificates referenced in `loadConfig` may be found.

`setCertificateVerifier`

```
void setCertificateVerifier(ContextType cType, CertificateVerifier certVerifier);
```

Set the `CertificateVerifier` used for the indicated `ContextType` role. All plug-in `Contexts` are created with default `CertificateVerifier` objects installed. Replacement `CertificateVerifiers` can be specified using this operation. This operation only affects new connections -- existing connections are left unchanged.

Parameters

`contextType`

The `Context(s)` in which to install the `CertificateVerifier`.

`certVerifier`

The `CertificateVerifier` to install.

See Also

`CertificateVerifier`.

setRSAKeys

```
void setRSAKeys(ContextType cType, ::Ice::ByteSeq privateKey,  
                 ::Ice::ByteSeq publicKey);
```

Set the RSA keys to be used by the plug-in when operating in the context mode specified by `ContextType`. This method only affects new connections -- existing connections are left unchanged.

Parameters

`contextType`

The `Context(s)` in which to set/replace the RSA keys.

`privateKey`

The RSA private key, in binary DER format.

`publicKey`

The RSA public key, in binary DER format.

setRSAKeysBase64

```
void setRSAKeysBase64(ContextType cType, string privateKey, string  
                      publicKey);
```

Set the RSA keys to be used by the plug-in when operating in the context mode specified by `ContextType`. This method only affects new connections -- existing connections are left unchanged.

Parameters

`contextType`

The Context(s) in which to set/replace the RSA keys.

`privateKey`

The RSA private key, in Base64-encoded binary DER format.

`publicKey`

The RSA public key, in Base64-encoded binary DER format.

B.131 `IceSSL::PrivateKeyException`

Overview

local exception `PrivateKeyException` extends `SslException`

A root exception class for all exceptions related to private keys.

Derived Exceptions

`PrivateKeyParseException`.

B.132 `IceSSL::PrivateKeyLoadException`

Overview

local exception `PrivateKeyLoadException` extends `ContextException`

Indicates that a problem occurred while loading a private key into a Context from either a memory buffer or from a file.

B.133 IceSSL::PrivateKeyParseException

Overview

Local exception `PrivateKeyParseException` extends `PrivateKeyException`

Indicates that IceSSL was unable to parse the provided private key into a form usable by the underlying SSL implementation.

B.134 IceSSL::ProtocolException

Overview

Local exception `ProtocolException` extends `ShutdownException`

Indicates that a problem occurred that violates the SSL protocol, causing the connection to be shutdown.

B.135 IceSSL::ShutdownException

Overview

Local exception `ShutdownException` extends `SslException`

This exception generally indicates that a problem occurred that caused the shutdown of an SSL connection.

Derived Exceptions

`CertificateVerificationException`, `ProtocolException`.

B.136 IceSSL::SslException

Overview

```
local exception SslException
```

This exception represents the base of all security related exceptions in Ice. It is a local exception because, usually, a problem with security precludes a proper secure connection over which to transmit exceptions. In addition, many exceptions would contain information that is of no use to external clients/servers.

Derived Exceptions

```
CertificateException, CertificateVerifierTypeException, ConfigParseException, ConfigurationLoadingException, ContextException, PrivateKeyException, ShutdownException.
```

message

```
string message;
```

Contains pertinent information from the security system to help explain the nature of the exception in greater detail. In some instances, it contains information from the underlying security implementation and/or debugging trace.

B.137 IceSSL::TrustedCertificateAddException

Overview

```
local exception TrustedCertificateAddException extends ContextException
```

An attempt to add a certificate to the Context's trusted certificate store has failed.

B.138 IceSSL::UnsupportedContextException

Overview

Local exception `UnsupportedContextException` extends `ContextException`

An attempt was made to call a method that references a `ContextType` that is not supported for that operation.

B.139 Glacier

Overview

`module Glacier`

Glacier is a firewall solution for Ice. Glacier authenticates and filters client requests and allows callbacks to the client in a secure fashion. In combination with IceSSL, Glacier provides a security solution that is both non-intrusive and easy to configure.

B.140 Glacier::CannotStartRouterException

Overview

`exception CannotStartRouterException`

This exception is raised if the router cannot be started.

`reason`

`string reason;`

Details as to why the router could not be started.

B.141 Glacier::NoSessionManagerException

Overview

```
exception NoSessionManagerException
```

This exception is raised if no SessionManager object has been configured.

B.142 Glacier::PermissionDeniedException

Overview

```
exception PermissionDeniedException
```

This exception is raised if router access is denied.

reason

```
string reason;
```

Details as to why access was denied.

B.143 Glacier::PermissionsVerifier

Overview

```
interface PermissionsVerifier
```

The Glacier router starter permissions verifier.

checkPermissions

```
bool checkPermissions(string userId, string password, out string  
reason);
```

Check whether user has permission to access router.

Parameters**userId**

The user id for which to check permissions.

password

The user's password.

reason

The reason access was denied.

Return Value

true if access is allowed, or false otherwise.

B.144 Glacier::Router

Overview

```
interface Router extends ::Ice::Router
```

The Glacier router interface.

createSession

```
Session* createSession() throws NoSessionManagerException;
```

Create a new session. The session is automatically shutdown when the Router terminates.

Return Value

A proxy to the new session.

Exceptions

```
NoSessionManagerException
```

if there is no configured SessionManager.

shutdown

```
void shutdown();
```

Shutdown the router.

B.145 Glacier::Session

Overview

`interface Session`

A session object, which is tied to the lifecycle of a Router.

See Also

`Router, SessionManager`.

`destroy`

`void destroy();`

Destroy the session. This is called automatically when the Router is destroyed.

B.146 Glacier::SessionManager

Overview

`interface SessionManager`

The session manager, which is responsible for managing Session objects. New session objects are created by the Router object.

See Also

`Session`.

`create`

`Session* create(string userId);`

Create a new session object.

Parameters**userId**

The user id for the session.

Return Value

A proxy to the newly created session.

B.147 Glacier::Starter

Overview**interface Starter**

The Glacier router starter.

startRouter

```
Router* startRouter(string userId, string password, out
    ::Ice::ByteSeq privateKey, out ::Ice::ByteSeq publicKey, out
    ::Ice::ByteSeq routerCert) throws PermissionDeniedException,
    CannotStartRouterException;
```

Start a new Glacier router. If the password for the given user id is incorrect, or if the user isn't allowed access, an `PermissionDeniedException` is raised. Otherwise a new router is started, and a proxy to that router is returned to the caller.

Parameters**userId**

The user id for which to check the password.

password

The password for the given user id.

privateKey

The RSA Private Key (DER encoded) for the client to use. (Only for SSL.)

publicKey

The RSA Public Key (DER encoded) for the client to use. (Only for SSL.)

routerCert

The trusted certificate of the router. (Only for SSL.)

Return Value

A proxy to the router that has been started.

Exceptions**PermissionDeniedException**

Raised if the password for the given user id is not correct or if the user isn't allowed access.

B.148 IceStorm

Overview**module IceStorm**

A messaging service with support for federation. In contrast to most other messaging or event services, IceStorm supports typed events, meaning that broadcasting a message over a federation is as easy as invoking a method on an interface.

LinkInfoSeq**sequence<LinkInfo> LinkInfoSeq;**

A sequence of `LinkInfo` objects.

Used By**Topic::getLinkInfoSeq.****QoS****dictionary<string, string> QoS;**

This dictionary represents Quality of service parameters.

Used By

Topic::subscribe.

See Also

Topic::subscribe.

TopicDict

dictionary<string, Topic*> TopicDict;

Mapping of topic name to topic proxy.

Used By

TopicManager::retrieveAll.

B.149 **IceStorm::LinkExists**

Overview

exception LinkExists

This exception indicates that an attempt was made to create a link that already exists.

name

string name;

The name of the linked topic.

B.150 **IceStorm::LinkInfo**

Overview

struct LinkInfo

Information on the topic links.

Used By

LinkInfoSeq.

cost

`int cost;`

The cost of traversing this link.

name

`string name;`

The name of the linked topic.

theTopic

`Topic* theTopic;`

The linked topic.

B.151 IceStorm::NoSuchLink

Overview

`exception NoSuchLink`

This exception indicates that an attempt was made to remove a link that does not exist.

name

`string name;`

The name of the link that does not exist.

B.152 IceStorm::NoSuchTopic

Overview

```
exception NoSuchTopic
```

This exception indicates that an attempt was made to retrieve a topic that does not exist.

name

```
string name;
```

The name of the topic that does not exist.

B.153 IceStorm::Topic

Overview

```
interface Topic
```

Publishers publish information on a particular topic. A topic logically represents a type.

See Also

[TopicManager](#).

destroy

```
void destroy();
```

Destroy the topic.

getLinkInfoSeq

```
LinkInfoSeq getLinkInfoSeq();
```

Retrieve information on the current links.

Return Value

A sequence of LinkInfo objects.

getName

```
string getName();
```

Get the name of this topic.

Return Value

The name of the topic.

See Also

`TopicManager::create`.

getPublisher

```
Object* getPublisher();
```

Get a proxy to a publisher object for this topic. To publish data to a topic, the publisher calls `getPublisher` and then casts to the topic type. An unchecked cast must be used on this proxy.

Return Value

A proxy to publish data on this topic.

link

```
void link(Topic* linkTo, int cost) throws LinkExists;
```

Create a link to the given topic. All events originating on this topic will also be sent to `link`.

Parameters**linkTo**

The topic to link to.

cost

The cost to the linked topic.

Exceptions**LinkExists**

Raised if a link to the same topic already exists.

subscribe

```
void subscribe(QoS theQoS, Object* subscriber);
```

Subscribe with the given QoS to this topic. If the given `subscribe` proxy has already been registered, it will be replaced.

Parameters**qos**

The quality of service parameters for this subscription.

subscriber

The subscriber's proxy.

See Also

`unsubscribe`.

unlink

```
void unlink(Topic* linkTo) throws NoSuchLink;
```

Destroy the link from this topic to the given topic `unlink`.

Parameters**link**

The topic to destroy the link to.

Exceptions**NoSuchLink**

Raised if a link to the topic does not exist.

unsubscribe

```
void unsubscribe(Object* subscriber);
```

Unsubscribe the given `unsubscribe`.

Parameters

`subscriber`

The proxy of an existing subscriber.

See Also

`subscribe`.

B.154 `IceStorm::TopicExists`

Overview

`exception TopicExists`

This exception indicates that an attempt was made to create a topic that already exists.

`name`

`string name;`

The name of the topic that already exists.

B.155 `IceStorm::TopicManager`

Overview

`interface TopicManager`

A topic manager manages topics, and subscribers to topics.

See Also

`Topic`.

create

```
Topic* create(string name) throws TopicExists;
```

Create a new topic. The topic name must be unique, otherwise `TopicExists` is raised.

Parameters

`name`

The name of the topic.

Return Value

A proxy to the topic instance.

Exceptions

`TopicExists`

Raised if a topic with the same name already exists.

retrieve

```
Topic* retrieve(string name) throws NoSuchTopic;
```

Retrieve a topic by name.

Parameters

`name`

The name of the topic.

Return Value

A proxy to the topic instance.

Exceptions

`NoSuchTopic`

Raised if the topic does not exist.

retrieveAll

```
TopicDict retrieveAll();
```

Retrieve all topics managed by this topic manager.

Return Value

A dictionary of string, topic proxy pairs.

B.156 `IcePatch`

Overview

```
module IcePatch
```

A patching service for software distributions. automates updating of individual files as well as complete directory hierarchies. Only files that have changed are downloaded to the client machine, using efficient compression algorithms.

`FileDescSeq`

```
sequence<FileDesc> FileDescSeq;
```

A sequence of file descriptions.

Used By

```
Directory::getContents.
```

B.157 `IcePatch::BusyException`

Overview

```
exception BusyException
```

This exception is raised when the server is busy performing housekeeping functions. The operation should be tried again later.

B.158 IcePatch::Directory

Overview

```
interface Directory extends File
```

The interface for a directory.

getContents

```
FileDescSeq getContents() throws FileAccessException,  
BusyException;
```

Obtain descriptions for the immediate children of the directory. This operation does not return the contents of subdirectories.

Return Value

The descriptions, each of which can be downcast to `RegularDesc` or `Directory-Desc`.

Exceptions

`FileAccessException`

Raised if a file or directory cannot be accessed.

`BusyException`

Raised if the server is busy.

getTotal

```
long getTotal(::Ice::ByteSeq md5) throws FileAccessException,  
BusyException;
```

Obtain the total number of bytes to be downloaded for a particular configuration.

Parameters

`md5`

The message digest of the client's existing configuration. If the given sequence is empty, then the return value represents the size of the entire directory, including all subdirectories. If the sequence is not empty, and the message

digest corresponds to a configuration that is known by the server, then the return value represents the number of bytes to be downloaded in order to update the client's directory to match the server's.

Return Value

The sum in bytes of the sizes of all BZ2 files to be downloaded for the configuration. If the given message digest does not match any known configurations, -1 is returned.

Exceptions

`FileAccessException`

Raised if a file or directory cannot be accessed.

`BusyException`

Raised if the server is busy.

B.159 `IcePatch::DirectoryDesc`

Overview

`class DirectoryDesc extends FileDesc`

Descriptive information for a directory.

`dir`

`Directory* dir;`

A proxy for the directory object.

B.160 `IcePatch::File`

Overview

`interface File`

The base interface for files and directories.

Derived Classes and Interfaces

Directory, Regular.

describe

`FileDesc describe() throws FileAccessException, BusyException;`

Obtain a description of the file or directory.

Return Value

The description, which can be downcast to `RegularDesc` or `DirectoryDesc`.

Exceptions

`FileAccessException`

Raised if the file or directory cannot be accessed.

`BusyException`

Raised if the server is busy.

B.161 IcePatch::FileAccessException

Overview

`exception FileAccessException`

This exception is raised if an error occurs while attempting to access a file or directory.

reason

`string reason;`

A description of the error.

B.162 IcePatch::FileDesc

Overview

```
class FileDesc
```

Descriptive information common to files and directories.

Derived Classes and Interfaces

DirectoryDesc, RegularDesc.

Used By

File::describe, FileDescSeq.

md5

```
::Ice::ByteSeq md5;
```

A sequence of 16 bytes containing the message digest of a file or directory computed using the MD5 algorithm. The message digest for a directory is computed using the names and message digests of each of the files and directories it contains.

B.163 IcePatch::Regular

Overview

```
interface Regular extends File
```

The interface for a regular file.

getBZ2

```
::Ice::ByteSeq getBZ2(int pos, int num) throws  
FileAccessException, BusyException;
```

Obtains a portion of the compressed file.

Parameters

`pos`

The starting position in bytes. The position must not exceed the size of the file.

`num`

The maximum number of bytes to return. This number must not exceed one megabyte.

Return Value

The requested bytes. The returned sequence may contain fewer bytes than requested if `getBZ2` exceeds the remaining bytes in the file.

Exceptions

`FileAccessException`

Raised if the file cannot be accessed.

`BusyException`

Raised if the server is busy.

getBZ2MD5

```
::Ice::ByteSeq getBZ2MD5(int size) throws FileAccessException,  
BusyException;
```

Obtain the message digest for a portion of the compressed file, allowing a client to determine whether a partially downloaded file is still valid.

Parameters

`size`

The number of bytes to use when computing the message digest.

Return Value

The number of bytes in the compressed representation of the file.

Exceptions

`FileAccessException`

Raised if the file cannot be accessed.

BusyException

Raised if the server is busy.

getBZ2Size

```
int getBZ2Size() throws FileAccessException, BusyException;
```

Obtain the compressed size of the file. compresses files using the BZIP2 algorithm.

Return Value

The number of bytes in the compressed representation of the file.

Exceptions**FileAccessException**

Raised if the file cannot be accessed.

BusyException

Raised if the server is busy.

B.164 [IcePatch::RegularDesc](#)**Overview**

```
class RegularDesc extends FileDesc
```

Descriptive information for a regular file.

reg

```
Regular* reg;
```

A proxy for the file object.

Appendix C

Properties

If not stated otherwise in the description of the individual properties, the default value for all properties is the empty string. If the property takes a numeric value, the empty string is interpreted as zero.

C.1 Ice Configuration Property

Ice.Config

Synopsis

```
--Ice.Config --Ice.Config=1 --Ice.Config=config_file
```

Description

This property must be set from the command line with the `--Ice.Config`, `--Ice.Config=1`, or `--Ice.Config=config_file` option.

If the `Ice.Config` property is empty or set to 1, the Ice run time examines the contents of the `ICE_CONFIG` environment variable to retrieve the pathname of a configuration file. Otherwise, `Ice.Config` must be set to the pathname of a configuration file. (Pathnames can be relative or absolute.) Further property values are read from the configuration file thus specified.

Configuration File Syntax

A configuration file contains a number of property settings, one setting per line. Property settings have one of the forms

property_name= # Set property to the empty string or zero

property_name=value # Assign value to property

The # character indicates a comment: the # character and anything following the # character on the same line are ignored. A line that has the # character as its first non-white space character is ignored in its entirety.

A configuration file is free-form: blank, tab, and newline characters serve as token delimiters and are otherwise ignored.

Any setting of the `Ice.Config` property inside the configuration file itself is ignored.

C.2 Ice Trace Properties

Ice.Trace.GC

Synopsis

`Ice.Trace.GC=num`

Description

The garbage collector trace level:

0	No garbage collector trace. (default)
1	Show the total number of instances collected, the total number of instances examined, the time spent in the collector in milliseconds, and the total number of runs of the collector.
2	Like 1, but also produces a trace message for each run of the collector.

Ice.Trace.Network

Synopsis

`Ice.Trace.Network=num`

Description

The network trace level:

0	No network trace. (default)
1	Trace connection establishment and closure.
2	Like 1, but more detailed.
3	Like 2, but also trace data transfer.

Ice.Trace.Protocol

Synopsis

`Ice.Trace.Protocol=num`

Description

The protocol trace level:

0	No protocol trace. (default)
1	Trace Ice protocol messages.

Ice.Trace.Retry

Synopsis

`Ice.Trace.Retry=num`

Description

The request retry trace level:

0	No request retry trace. (default)
1	Trace Ice operation call retries.
2	Also trace Ice endpoint usage.

Ice.Trace.Slicing**Synopsis**

`Ice.Trace.Slicing=num`

Description

The slicing trace level:

0	No trace of slicing activity. (default)
1	Trace all exception and class types that are unknown to the receiver and therefore sliced.

C.3 Ice Warning Properties

Ice.Warn.Connections**Synopsis**

`Ice.Warn.Connections=num`

Description

If `num` is set to a value larger than zero, Ice applications will print warning messages for certain exceptional conditions in connections. The default value is 0.

Ice.Warn.Datagrams

Synopsis

`Ice.Warn.Datagrams=num`

Description

If *num* is set to a value larger than zero, servers will print a warning message if they receive a datagram that exceeds the servers' receive buffer size. (Note that this condition is not detected by all UDP implementations -- some implementations silently drop received datagrams that are too large.) The default value is 0.

Ice.Warn.Dispatch

Synopsis

`Ice.Warn.Dispatch=num`

Description

If *num* is set to a value larger than zero, Ice applications will print warning messages for certain exceptions that are raised while an incoming request is dispatched.

0	No warnings.
1	Print warnings for unexpected <code>Ice::LocalException</code> , <code>Ice::UserException</code> , C++ exceptions, and Java runtime exceptions. (default)
2	Like 1, but also issue warnings for <code>Ice::ObjectNotExistException</code> , <code>Ice::FacetNotExistException</code> , and <code>Ice::OperationNotExistException</code> .

Ice.Warn.AMICallback

Synopsis

`Ice.Warn.AMICallback=num`

Description

If *num* is set to a value larger than zero, warnings are printed if an AMI callback raises an exception. The default value is 1.

Ice.WarnLeaks**Synopsis**

`Ice.WarnLeaks=num`

Description

If *num* is set to a value larger than zero, the `Ice::Communicator` destructor will print a warning if some other Ice-related C++ objects are still in memory. The default value is 1. (C++ only.)

C.4 Ice Object Adapter Properties

name.AdapterId**Synopsis**

`name.AdapterId=id`

Description

Specifies an identifier for the object adapter with the name *name*. This identifier must be unique among all object adapters using the same locator instance. If a locator proxy is defined using `name.Locator` or `Ice.Default.Locator`, this object adapter will set its endpoints with the locator registry upon activation.

name.Endpoints**Synopsis**

`name.Endpoints=endpoints`

Description

Sets the endpoints for the object adapter *name* to *endpoints*.

name.Locator

Synopsis

name.Locator=locator

Description

Specifies a locator for the object adapter with the name *name*. The value is a stringified proxy to the Ice locator interface.

name.RegisterProcess

Synopsis

name.RegisterProcess=num

Description

If *num* is set to a value larger than zero, the object adapter with the name *name* registers the server with the locator registry. Registration occurs upon the object adapter's initial activation, during which the object adapter creates a servant implementing the `Ice::Process` interface, adds the servant using a UUID, and registers its proxy with the locator registry using the value of the `Ice.ServerId` property for the server id. The servant implements the `shutdown` operation by invoking `shutdown` on the object adapter's communicator.

It is important for a server to be registered with the locator registry so that services such as IcePack can request a graceful shutdown when necessary. If a server is not registered, then platform-specific techniques are used to request a shutdown, and these techniques are not always effective (especially on Windows platforms).

Only one object adapter in a server should register with a locator registry.

The `Ice::Process` servant represents a potential target for denial-of-service attacks. The object adapter uses a UUID to make the proxy more difficult to guess, but the object adapter should be configured with secure endpoints if the server operates in a potentially hostile environment. Alternatively, a dedicated object adapter can be created specifically to provide a restricted access point for services such as IcePack.

name.Router**Synopsis**

name.Router=router

Description

Specifies a router for the object adapter with the name *name*. The value is a stringified proxy to the Glacier router control interface. Defining a router allows the object adapter to receive callbacks from the router over outgoing connections from this process to the router, thereby avoiding the need for the router to establish a connection back to the object adapter.

A router can only be assigned to one object adapter. Specifying the same router for more than one object adapter will result in undefined behavior. The default value is no router.

Ice.PrintAdapterReady**Synopsis**

Ice.PrintAdapterReady=num

Description

If *num* is set to a value larger than zero, an object adapter prints "*adapter_name* ready" on standard output after initialization is complete. This is useful for scripts that need to wait until an object adapter is ready to be used.

C.5 Ice Plug-in Properties

Ice.Plugin.name**Synopsis**

Ice.Plugin.name=entry_point [args]

Description

Defines a plug-in to be installed during communicator initialization.

In C++, *entry_point* has the form `basename[,version]:function`. The `basename` and optional `version` components are used to construct the name of a DLL or shared library. If no version is supplied, the Ice version is used. The `function` component is the name of a function with C linkage.

For example, the entry point `MyPlugin,2.3:create` would imply a shared library name of `libMyPlugin.so.2.3` on Unix and `MyPlugin23.dll` on Windows. Furthermore, if Ice is built on Windows with debugging, a `d` will be automatically appended to the version (e.g., `MyPlugin23d.dll`).

In Java, *entry_point* is the name of a class.

C.6 Ice Thread Pool Properties

name.Size

Synopsis

`name.Size=num`

Description

name is the name of the thread pool. The name of the client-side thread pool is `Ice.ThreadPool.Client`, the name of the default server-side thread pool is `Ice.ThreadPool.Server`. In addition, individual object adapters can have separate thread pools. In this case, the name of the thread pool is `adapter_name.ThreadPool`. Having a separate thread pool for an object adapter is useful to ensure that a minimum number of threads is available for dispatching requests on certain Ice objects, in order to avoid deadlocks because of thread starvation.

num is the initial and also minimum number of threads in the thread pool. The default is one for `Ice.ThreadPool.Client` and `Ice.ThreadPool.Server`, and zero for object adapter thread pools, meaning that by default, object adapters use the default server-side thread pool.

Multiple threads for the client side thread pool are only required for nested AMI invocations. If AMI is not used, or AMI calls are not nested (i.e., AMI callbacks do not call any other methods on Ice objects), then there is no need to set the number of threads in the client thread pool to a value larger than one.

name.SizeMax**Synopsis***name.SizeMax=num***Description**

num is the maximum number of threads for the thread pool *name*. Thread pools in Ice can grow and shrink dynamically, based on an average load factor. Thread pools do not grow larger than the parameter specified by `SizeMax`, and they do not shrink to a number of threads smaller than the value specified by `Size`.

The default value for `SizeMax` is the value of `Size`, meaning that by default, thread pools do not grow dynamically.

name.SizeWarn**Synopsis***name.SizeWarn=num***Description**

Whenever *num* threads are active in a thread pool, a "low on threads" warning is printed. The default value for `SizeWarn` is 80% of the value specified by `SizeMax`.

C.7 Ice Default and Override Properties

`Ice.Default.Protocol`**Synopsis**`Ice.Default.Protocol=protocol`**Description**

Sets the protocol that is being used if an endpoint uses `default` as the protocol specification. The default value is `tcp`.

Ice.Default.Host

Synopsis

`Ice.Default.Host=host`

Description

If an endpoint is specified without a host name (i.e., without a `-h host` option), the `host` value from this property is used instead. The default value is the local host name.

Ice.Default.Router

Synopsis

`Ice.Default.Router=router`

Description

Specifies the default router for all proxies. The value is a stringified proxy to the Glacier router control interface. The default router can be overridden on a proxy using the `ice_router()` operation. The default value is no router.

Ice.Default.Locator

Synopsis

`Ice.Default.Locator=locator`

Description

Specifies a default locator for all proxies and object adapters. The value is a stringified proxy to the IcePack locator interface. The default locator can be overridden on a proxy using the `ice_locator()` operation. The default value is no locator.

The IcePack locator's object identity is `IcePack/Locator`. It is listening on the IcePack client endpoints. For example if `IcePack.Registry.Client.Endpoints` is set to `tcp -p 12000 -h localhost`, the stringified proxy for the IcePack locator will be `IcePack/Locator:tcp -p 12000 -h localhost`.

Ice.Override.Timeout

Synopsis

`Ice.Override.Timeout=num`

Description

If set, this property overrides timeout settings in all endpoints. *num* is the timeout value in milliseconds, or -1 for no timeout.

Ice.Override.ConnectTimeout

Synopsis

`Ice.Override.ConnectTimeout=num`

Description

This property overrides timeout settings used to establish connections. *num* is the timeout value in milliseconds, or -1 for no timeout. If this property is not set, then `Ice.Override.Timeout` is used.

Ice.Override.Compress

Synopsis

`Ice.Override.Compress=num`

Description

If set, this property overrides compression settings in all proxies. If *num* is set to a value larger than zero, compression is enabled. If zero, compression is disabled.

C.8 Ice Miscellaneous Properties

Ice.GC.Interval

Synopsis

`Ice.GC.Interval=num`

Description

This property determines the frequency with which the class garbage collector runs. If the interval is set to zero (the default), no collector thread is created. Otherwise, the collector thread runs every *num* seconds.

Ice.RetryIntervals

Synopsis

`Ice.RetryIntervals=num [num ...]`

Description

This property defines the number of times an operation is re-tried and the delay between each retry. For example, if the property is set to *0 100 500*, the operation will be re-tried 3 times: immediately re-tried upon the first failure, after waiting 100 (ms) upon the second failure, after waiting 500 (ms) upon the third failure. The default value is to retry once immediately (0). If set to -1, no retry will occur.

Ice.MessageSizeMax

Synopsis

`Ice.MessageSizeMax=num`

Description

This property controls the maximum size (in kilobytes) of a protocol message that will be accepted or sent by the Ice run time. The size includes the size of the Ice protocol header. Messages larger than this size cause a [MemoryLimitException]. The default size is 1024 (1 Megabyte). Settings with a value less than 1 are ignored.

This property adjusts the value of `Ice.UDP.RcvSize` and `Ice.UDP.SndSize`, that is, if `Ice.UDP.RcvSize` or `Ice.UDP.SndSize` are larger than `Ice.MessageSizeMax * 1024 + 28`, they are adjusted to `Ice.MessageSizeMax * 1024 + 28`.

Ice.ChangeUser

Synopsis

`Ice.ChangeUser=user`

Description

If set, Ice will change the user and group id to the respective ids of *user* in `/etc/passwd`. This only works if the Ice application is executed by the super-user. (Unix only.)

Ice.ConnectionIdleTime

Synopsis

`Ice.ConnectionIdleTime=num`

Description

If *num* is set to a value larger than zero, ACM (Active Connection Management) is enabled. This means that connections are automatically closed after they have been idle for *num* seconds. This is transparent to user code, i.e., closed connections are automatically reestablished in case they are needed again. The default value is 60, meaning that idle connections are automatically closed after one minute.

Ice.MonitorConnections

Synopsis

`Ice.MonitorConnections=num`

Description

If *num* is set to a value larger than zero, Ice will start a thread that monitors connections. This thread is responsible for shutting down idle connections (see `Ice.ConnectionIdleTime`), as well as for enforcing AMI timeouts. The default value is the value of `Ice.ConnectionIdleTime`.

Ice.PrintProcessId

Synopsis

`Ice.PrintProcessId=num`

Description

If *num* is set to a value larger than zero, the process ID is printed on standard output upon startup.

Ice.ProgramName

Synopsis

`Ice.ProgramName=name`

Description

name is the program name, which is set automatically from `argv[0]` (C++) and from `AppDomain.CurrentDomain.FriendlyName` (C#) during initialization. (For Java, `Ice.ProgramName` is initialized to the empty string.) For all languages, the default name can be overridden by setting this property.

Ice.ServerId

Synopsis

`Ice.ServerId=id`

Description

The value *id* is used as the server id when an object adapter registers the server with the locator registry. Refer to the description of the object adapter property `adapter.RegisterProcess` for more information.

Ice.ServerIdleTime

Synopsis

`Ice.ServerIdleTime=num`

Description

If *num* is set to a value larger than zero, Ice will automatically call `Communicator::shutdown` after the Communicator has been idle for *num* seconds. This will shut down the Communicator's server side, and all threads waiting in `Communicator::waitForShutdown` will return. After that, a server will typically do some cleanup work, and then exit. The default value is zero, meaning that the server will not shut down automatically.

Ice.UseEventLog**Synopsis**

`Ice.UseEventLog=num`

Description

If *num* is set to a value larger than zero, a special logger is installed that logs to the Windows Event Log instead of standard error. The event source name is the value of `Ice.ProgramName`. (Windows 2000/XP only.)

Ice.UseSyslog**Synopsis**

`Ice.UseSyslog=num`

Description

If *num* is set to a value larger than zero, a special logger is installed that logs to the syslog facility instead of standard error. The identifier for syslog is the value of `Ice.ProgramName`. (Unix only.)

Ice.Logger.Timestamp**Synopsis**

`Ice.Logger.Timestamp=num`

Description

If *num* is set to a value larger than zero, the output of the default logger will include timestamps.

Ice.NullHandleAbort

Synopsis

`Ice.NullHandleAbort=num`

Description

If *num* is set to a value larger than zero, invoking an operation using a null smart pointer (i.e., a handle) causes the program to abort immediately instead of raising `IceUtil::NullHandleException`. (C++ only.)

Ice.Nohup

Synopsis

`Ice.Nohup=num`

Description

If *num* is set to a value larger than zero, the C++ classes `Ice::Application` and `Ice::Service` ignore SIGHUP (for UNIX) and CTRL_LOGOFF_EVENT (for Windows). As a result, an application that sets `Ice.Nohup` continues to run if the user that started the application logs off. The default value for `Ice::Application` is 0, and the default value for `Ice::Service` is 1 (C++ only.)

Ice.UDP.RcvSize, Ice.UDP.SndSize

Synopsis

`Ice.UDP.RcvSize=num` `Ice.UDP.SndSize=num`

Description

These properties set the UDP receive and send buffer sizes to the specified value in bytes. Ice messages larger than *num* - 28 bytes cause a `DatagramLimitException`.

tion. The default value depends on the configuration of the local UDP stack. (Common default values are 65535 and 8192 bytes.)

The OS may impose lower and upper limits on the receive and send buffer sizes or otherwise adjust the buffer sizes. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

Settings of these properties less than 28 are ignored.

Note that, on many operating systems, it is possible to set buffer sizes greater than 65535. Such settings do not change the hard limit of 65507 bytes for the payload of a UDP packet, but merely affect how much data can be buffered by the kernel.

Settings less than 65535 limit the size of Ice datagrams as well as adjust the kernel buffer sizes.

If `Ice.MessageSizeMax` is set and `Ice.MessageSizeMax * 1024 + 28` is smaller than the UDP receive or send buffer size, the corresponding UDP buffer size is reduced to `Ice.MessageSizeMax * 1024 + 28`.

C.9 IceSSL Properties

IceSSL.Trace.Security

Synopsis

`IceSSL.Trace.Security=num`

Description

The SSL plug-in trace level:

0	No security trace. (default)
1	Trace security warnings.
2	Like 1, but more verbose, including warnings during configuration file parsing.

IceSSL.Client.CertPath, IceSSL.Server.CertPath

Synopsis

`IceSSL.Client.CertPath=path IceSSL.Server.CertPath=path`

Description

Defines the path (relative or absolute) where the SSL plug-in can find PEM format certificate files (RSA and DSA) and Diffie-Hellman group parameter files.

In the case that the `IceSSL.Client.Config` or `IceSSL.Server.Config` is a relative path, it will be relative to the value of `IceSSL.Client.CertPath` or `IceSSL.Server.CertPath`.

If not specified, the application will use the current working directory as the certificate path.

IceSSL.Client.Config, IceSSL.Server.Config

Synopsis

`IceSSL.Client.Config=config_file IceSSL.Server.Config=config_file`

Description

Defines the XML-based configuration file from which the SSL plug-in will load initialization information and certificates. If the property contains a relative path, the path will be interpreted relative to the certificate path defined by `IceSSL.Client.CertPath` or `IceSSL.Server.CertPath`.

The XML parser looks for the DTD file in the same directory in which it finds the XML configuration file.

Depending on whether the application is running in client mode, server mode or both modes, a valid value for one or both of these parameters must be specified for the proper operation of the IceSSL plug-in.

IceSSL.Client.Handshake.Retries

Synopsis

`IceSSL.Client.Handshake.Retries=num`

Description

IceSSL clients attempt to perform an entire SSL handshake in the connection phase. When attempting this handshake, it is possible that the client will timeout waiting for a response from the server. This property specifies the number of handshake retries the client attempts before throwing a `Ice::ConnectionFailedException`.

If not specified, the default value for this property is 10 retries.

**`IceSSL.Client.Passphrase.Retries`,
`IceSSL.Server.Passphrase.Retries`****Synopsis**

`IceSSL.Client.Passphrase.Retries=num`
`IceSSL.Server.Passphrase.Retries=num`

Description

When IceSSL is directed to use a private key in a PEM file that has been encrypted, a prompt is displayed `Enter PEM pass phrase:`. If the passphrase is entered incorrectly, these properties determine how many retries the user will be allowed before IceSSL shuts down.

If not specified, the default value for these properties is 5 retries.

**`IceSSL.ServerOverrides.RSA.PrivateKey`,
`IceSSL.ServerOverrides.RSA.Certificate`****Synopsis**

`IceSSL.ServerOverrides.RSA.PrivateKey=Base64 encoded DER string`
`IceSSL.ServerOverrides.RSA.Certificate=Base64 encoded DER string`

Description

These properties override the RSA private key and public key (certificate) specified in the config file (`IceSSL.Server.Config`) for the Server context. The value must be the DER representation of the private and public keys, base64 encoded.

There are no default values for these properties.

`IceSSL.ServerOverrides.DSA.PrivateKey`,

`IceSSL.Server.Overrides.DSA.Certificate`

Synopsis

`IceSSL.Server.Overrides.DSA.PrivateKey=Base64 encoded DER string`
`IceSSL.Server.Overrides.DSA.Certificate=Base64 encoded DER string`

Description

These properties override the DSA private key and public key (certificate) specified in the config file (`IceSSL.Server.Config`) for the Server context. The value must be the DER representation of the private and public keys, base64 encoded.

There are no default values for these properties.

`IceSSL.Client.Overrides.RSA.PrivateKey,` `IceSSL.Client.Overrides.RSA.Certificate`

Synopsis

`IceSSL.Client.Overrides.RSA.PrivateKey=Base64 encoded DER string`
`IceSSL.Client.Overrides.RSA.Certificate=Base64 encoded DER string`

Description

These properties provides a method by which the RSA private key and public key (certificate) used by the Client context may be overridden from those specified in the config file (specified in `IceSSL.Client.Config`). The value must be the DER representation of the private and public keys, base64 encoded.

There are no default values for these properties.

`IceSSL.Client.Overrides.DSA.PrivateKey,` `IceSSL.Client.Overrides.DSA.Certificate`

Synopsis

`IceSSL.Client.Overrides.DSA.PrivateKey=Base64 encoded DER string`
`IceSSL.Client.Overrides.DSA.Certificate=Base64 encoded DER string`

Description

These properties override the RSA private key and public key (certificate) specified in the config file (`IceSSL.Client.Config`) for the Client context. The value must be the DER representation of the private and public keys, base64 encoded.

There are no default values for these properties.

`IceSSL.Client.Overrides.CACertificate,` `IceSSL.Server.Overrides.CACertificate`

Synopsis

`IceSSL.Client.Overrides.CACertificate=Base64 encoded DER string`
`IceSSL.Server.Overrides.CACertificate=Base64 encoded DER string`

Description

These properties override any trusted Certificate Authority (CA) certificates specified in `IceSSL.Server.Config` or `IceSSL.Client.Config`. The new certificate is represented as the base64 encoding of the DER binary representation of the certificate.

There are no default values for these properties.

`IceSSL.Client.IgnoreValidPeriod,` `IceSSL.Server.IgnoreValidPeriod`

Synopsis

`IceSSL.Client.IgnoreValidPeriod=num`
`IceSSL.Server.IgnoreValidPeriod=num`

Description

These properties will cause the default certificate verifier to ignore the certificate validity period on peer certificates if set to 1. The default value for these properties is 0, meaning that the certificate validity period is not ignored.

C.10 IceBox Properties

IceBox.ServiceManager.Endpoints

Synopsis

`IceBox.ServiceManager.Endpoints=endpoints`

Description

Defines the endpoints of the IceBox service manager interface. The service manager endpoints must be accessible to the IceBox administration tool to shutdown the IceBox server.

IceBox.ServiceManager.Identity

Synopsis

`IceBox.ServiceManager.Identity=identity`

Description

The identity of the service manager interface. If not specified the default value ServiceManager is used.

IceBox.PrintServicesReady

Synopsis

`IceBox.PrintServicesReady=token`

Description

The service manager will print "*token* ready" on standard output after initialization of all the services is done. This is useful for scripts that wish to wait until all services are ready to be used.

IceBox.LoadOrder

Synopsis

`IceBox.LoadOrder=names`

Description

Determines the order in which services are loaded. The service manager loads the services in the order they appear in *names*, where each service name is separated by a comma or whitespace. Any services not mentioned in *names* are loaded afterward, in an undefined order.

IceBox.Service.*name*

Synopsis

`IceBox.Service.name=entry_point [args]`

Description

Defines a service to be loaded during IceBox initialization.

In C++, *entry_point* has the form `library:symbol`. The `library` component is the name of a shared library or DLL. The `symbol` component is the name of a factory function used to create the service.

In Java, *entry_point* is the name of the service implementation class.

IceBox.DBEnvName.*name*

Synopsis

`IceBox.DBEnvName.name=db`

Description

Defines the database environment directory for the Freeze service with the name *name*.

IceBox.UseSharedCommunicator.name

Synopsis

`IceBox.UseSharedCommunicator.name=num`

Description

If *num* is set to a value larger than zero, the service manager will supply the service with the name *name* a communicator which might be shared by other services.

C.11 IcePack Properties

IcePack.Registry.Client.Endpoints

Synopsis

`IcePack.Registry.Client.Endpoints=endpoints`

Description

Defines the endpoints of the IcePack client interface. The client endpoints must be accessible to Ice clients that are using IcePack to locate objects (see `Ice.Default.Locator`).

IcePack.Registry.Server.Endpoints

Synopsis

`IcePack.Registry.Server.Endpoints=endpoints`

Description

Defines the endpoints of the IcePack server interface. The server endpoints must be accessible to Ice servers that are using IcePack to register their object adapter endpoints.

IcePack.Registry.Admin.Endpoints

Synopsis

`IcePack.Registry.Admin.Endpoints=endpoints`

Description

Defines the optional administrative endpoints of the IcePack admin interface. The administrative endpoints must be accessible to clients which are using the IcePack administrative interface, such as the IcePack administrative tool.

Allowing access to the IcePack admin interface is a security risk! If this property is not defined, the admin interface will be disabled.

IcePack.Registry.Internal.Endpoints

Synopsis

`IcePack.Registry.Internal.Endpoints=endpoints`

Description

Defines the endpoints of the IcePack internal interface. The internal endpoints must be accessible to IcePack nodes. Nodes use this interface to communicate with the registry.

IcePack.Registry.Data

Synopsis

`IcePack.Registry.Data=path`

Description

Defines the path of the IcePack registry data directory.

IcePack.Registry.DynamicRegistration

Synopsis

`IcePack.Registry.DynamicRegistration=num`

Description

If *num* is set to a value larger than zero, the locator registry will allow Ice servers to set endpoints for object adapters which have not been previously registered.

IcePack.Registry.Trace.ServerRegistry

Synopsis

`IcePack.Registry.Trace.ServerRegistry=num`

Description

The server registry trace level:

0	No server registry trace. (default)
1	Trace server registration, removal.

IcePack.Registry.Trace.AdapterRegistry

Synopsis

`IcePack.Registry.Trace.AdapterRegistry=num`

Description

The object adapter registry trace level:

0	No object adapter registry trace. (default)
1	Trace object adapter registration, removal.

IcePack.Registry.Trace.NodeRegistry

Synopsis

`IcePack.Registry.Trace.NodeRegistry=num`

Description

The node registry trace level:

0	No node registry trace. (default)
1	Trace node registration, removal.

IcePack.Registry.Trace.ObjectRegistry**Synopsis**

`IcePack.Registry.Trace.ObjectRegistry=num`

Description

The object registry trace level:

0	No object registry trace. (default)
1	Trace object registration, removal.

IcePack.Node.Endpoints**Synopsis**

`IcePack.Node.Endpoints=endpoints`

Description

Defines the endpoints of the IcePack node interface. The node endpoints must be accessible to the IcePack registry. The registry uses this interface to communicate with the node.

IcePack.Node.Name**Synopsis**

`IcePack.Node.Name=name`

Description

Defines the name of the IcePack node. All nodes using the same registry must have unique names. The node will refuse to start if there is a node with the same name already running.

The default value is the hostname as returned by `gethostname()`.

IcePack.Node.Data

Synopsis

`IcePack.Node.Data=path`

Description

Defines the path of the IcePack node data directory. The node will create db and servers subdirectories in this directory if they do not already exist. The db directory contains the node database. The servers directory contains configuration data for each deployed server.

IcePack.Node.Output

Synopsis

`IcePack.Node.Output=path`

Description

Defines the path of the IcePack node output directory. If set, the node will redirect the stdout and stderr output of the started servers to files named *server.out* and *server.err* in this directory. Otherwise, the started servers share the stdout and stderr of the IcePack node.

IcePack.Node.PropertiesOverride

Synopsis

`IcePack.Node.PropertiesOverride=overrides`

Description

Defines a list of properties which override the properties defined in server deployment descriptors. For example, in some cases it is desirable to set the property `Ice.Default.Host` for servers, but not in server deployment descriptors. The property definitions should be separated by white space.

`IcePack.Node.RedirectErrToOut`**Synopsis**

`IcePack.Node.RedirectErrToOut=num`

Description

If *num* is set to a value larger than zero, the stderr of each started server is redirected to the server's stdout.

`IcePack.Node.WaitTime`**Synopsis**

`IcePack.Node.WaitTime=num`

Description

Defines the interval in seconds that IcePack will wait for server activation and deactivation.

If a server is automatically activated and does not register its object adapter endpoints within this time interval, the node will assume there is a problem with the server and return an empty set of endpoints to the client.

If a server is being gracefully deactivated and IcePack does not detect the server deactivation during this time interval, IcePack will kill the server.

The default value is 60 seconds.

`IcePack.Node.Collect Registry`**Synopsis**

`IcePack.Node.Collect Registry=num`

Description

If *num* is set to a value larger than zero, the node will also collocate the IcePack registry.

The collocated registry is configured with the same properties as the standalone IcePack registry.

IcePack.Node.PrintServersReady

Synopsis

`IcePack.Node.PrintServersReady=token`

Description

The IcePack node will print "*token* ready" on standard output after all the servers managed by the node are ready. This is useful for scripts that wish to wait until all servers are ready to be used.

IcePack.Node.Trace.Server

Synopsis

`IcePack.Node.Trace.Server=num`

Description

The server trace level:

0	No server trace. (default)
1	Trace server addition, removal.
2	Like 1, but more verbose, including server activation and deactivation and more diagnostic messages.
3	Like 2, but more verbose, including server transitional state change (activating and deactivating).

IcePack.Node.Trace.Adapter

Synopsis

`IcePack.Node.Trace.Adapter=num`

Description

The object adapter trace level:

0	No object adapter trace. (default)
1	Trace object adapter addition, removal.
2	Like 1, but more verbose, including object adapter activation and deactivation and more diagnostic messages.
3	Like 2, but more verbose, including object adapter transitional state change (e.g., 'waiting for activation').

IcePack.Node.Trace.Activator

Synopsis

`IcePack.Node.Trace.Activator=num`

Description

The activator trace level:

0	No activator trace. (default)
1	Trace process activation, termination.
2	Like 1, but more verbose, including process signaling and more diagnostic messages on process activation.
3	Like 2, but more verbose, including more diagnostic messages on process activation (e.g., path, working directory and arguments of the activated process).

C.12 IceStorm Properties

IceStorm.TopicManager.Endpoints

Synopsis

`IceStorm.TopicManager.Endpoints=endpoints`

Description

Defines the endpoints for the IceStorm topic manager and topic objects.

IceStorm.Publish.Endpoints

Synopsis

`IceStorm.Publish.Endpoints=endpoints`

Description

Defines the endpoints for the IceStorm publisher objects.

IceStorm.Trace.TopicManager

Synopsis

`IceStorm.Trace.TopicManager=num`

Description

The topic manager trace level:

0	No topic manager trace. (default)
1	Trace topic creation.

IceStorm.Trace.Topic

Synopsis

`IceStorm.Trace.Topic=num`

Description

The topic trace level:

0	No topic trace. (default)
1	Trace topic links, subscription, and unsubscription.
2	Like 1, but more verbose, including QoS information, and other diagnostic information.

IceStorm.Trace.Flush**Synopsis**

`IceStorm.Trace.Flush=num`

Description

Trace information on the thread that flushes batch reliability events to subscribers:

0	No flush trace. (default)
1	Trace each flush.

IceStorm.Trace.Subscriber**Synopsis**

`IceStorm.Trace.Subscriber=num`

Description

The subscriber trace level:

0	No subscriber trace. (default)
1	Trace topic diagnostic information on subscription and unsubscription.

IceStorm.Flush.Timeout

Synopsis

`IceStorm.Flush.Timeout=num`

Description

Defines the interval in milliseconds that batch reliability events are sent to subscribers. The default is 1000 ms. The minimum value of this property is 100 ms.

IceStorm.TopicManager.Proxy

Synopsis

`IceStorm.TopicManager.Proxy=proxy`

Description

Defines the proxy for the IceStorm topic manager. This property is used by the IceStorm administration tool, and may also be used by applications.

C.13 Glacier Router Properties

Glacier.Router.Endpoints, Glacier.Router.Client.Endpoints, Glacier.Router.Server.Endpoints

Synopsis

`Glacier.Router.Endpoints=endpoints`
`Glacier.Router.Client.Endpoints=endpoints`
`Glacier.Router.Server.Endpoints=endpoints`

Description

Defines the endpoints of the Glacier router control interface, the client interface, and the server interface. The router endpoints and the client endpoints must be accessible to Glacier clients from which the router forwards requests, and to which the router sends callbacks. The server endpoints must be accessible to

Glacier servers to which the router forwards requests, and from which the router accepts callbacks for the client.

Glacier.Router.Identity

Synopsis

`Glacier.Router.Identity=identity`

Description

The identity of the router control interface. If not specified, the default value `router` is used.

Glacier.Router.PrintProxyOnFd

Synopsis

`Glacier.Router.PrintProxyOnFd=fd`

Description

If set, print the stringified router proxy on the filedescriptor `fd`, and close this filedescriptor. (Unix only.)

This operation is intended to be used by the Glacier router starter only. It should not be set manually.

Glacier.Router.Trace.Client

Synopsis

`Glacier.Router.Trace.Client=num`

Description

The client interface trace level:

0	No client interface trace. (default)
1	Trace exceptions during request forwarding from the client to the server.

2	Also trace detailed forward routing information from the client to the server.
---	--

Glacier.Router.Trace.Server

Synopsis

`Glacier.Router.Trace.Server=num`

Description

The server interface trace level:

0	No server interface trace. (default)
1	Trace exceptions during callbacks from the server back to the client.
2	Also trace detailed reverse routing information for callbacks from the server to the client.

Glacier.Router.Trace.RoutingTable

Synopsis

`Glacier.Router.Trace.RoutingTable=num`

Description

The routing table trace level:

0	No routing table trace. (default)
1	Trace additions to the Glacier routing table.

Glacier.Router.Trace.Throttle

Synopsis

`Glacier.Router.Trace.Throttle=num`

Description

The request throttle trace level:

0	No request throttle tracing. (default)
1	Trace throttled requests.

Glacier.Router.Client.ForwardContext, Glacier.Router.Server.ForwardContext

Synopsis

`Glacier.Router.Client.ForwardContext=num`
`Glacier.Router.Server.ForwardContext=num`

Description

If *num* is set to a value larger than zero, the context parameter is forwarded unmodified as received from the client or server, respectively. Otherwise an empty context is forwarded. Default is no context forwarding.

Glacier.Router.Client.SleepTime, Glacier.Router.Server.SleepTime

Synopsis

`Glacier.Router.Client.SleepTime=num`
`Glacier.Router.Server.SleepTime=num`

Description

num is the sleep time (delay) in milliseconds after a request or message batch has been forwarded. The default value is zero, meaning no delay. Setting these values avoids message flooding. This is useful to avoid denial-of-service attacks, or to allow a minimum time for collecting messages for batching.

Glacier.Router.Client.Throttle.Twoways, Glacier.Router.Server.Throttle.Twoways

Synopsis

`Glacier.Router.Client.Throttle.Twoways=num`
`Glacier.Router.Server.Throttle.Twoways=num`

Description

num is the maximum number of two-way requests that the router will forward simultaneously. The default is zero, meaning no limit. Enabling two-way message throttling is useful to prevent a router from consuming too many resources of backend services.

Glacier.Router.SessionManager

Synopsis

`Glacier.Router.SessionManager=proxy`

Description

A stringified proxy to a session manager. If not specified, it is not possible to use the `Router::createSession()` method.

Glacier.Router.UserId

Synopsis

`Glacier.Router.UserId=name`

Description

The authenticated user id. This is usually passed from the Glacier router starter. The user id is used as an argument to `Router::createSession()`.

Glacier.Router.AllowCategories

Synopsis

`Glacier.Router.AllowCategories=list`

Description

A white space separated list of categories. If this property is set, then requests are only permitted to Ice objects with an identity that matches one of the categories from this list.

Glacier.Router.AcceptCert**Synopsis**

`Glacier.Router.AcceptCert=base64 encoded certificate string`

Description

A base64 encoded certificate (which can be obtained by calling `certToBase64()` on an existing `IceSSL::RSAKeyPair`). The Glacier Router will use this certificate, when in SSL mode, to restrict those clients that may connect to it. Only clients that use this certificate may connect, others will be rejected.

C.14 Glacier Router Starter Properties

Glacier.Starter.Endpoints**Synopsis**

`Glacier.Starter.Endpoints=endpoints`

Description

Defines the endpoints of the Glacier router starter. (Unix only.)

Glacier.Starter.PasswordVerifier**Synopsis**

`Glacier.Starter.PasswordVerifier=proxy`

Description

If set, the specified password verifier will be used. If not set, a built-in crypt-based password verifier will be used.

Glacier.Starter.CryptPasswords

Synopsis

`Glacier.Starter.CryptPasswords=file`

Description

The pathname of the file that contains user-id / password pairs, with the passwords being encrypted by the `crypt` algorithm. The default pathname is "passwords". This file is only used for the built-in `crypt`-based password verifier, meaning that the property is ignored if `Glacier.Starter.PasswordVerifier` is set.

Glacier.Starter.RouterPath

Synopsis

`Glacier.Starter.RouterPath=path`

Description

Sets the path of the Glacier router executable to be started. The default is `glacier`. (Unix only.)

Glacier.Starter.PropertiesOverride

Synopsis

`Glacier.Starter.PropertiesOverride=overrides`

Description

By default, the Glacier router starter starts new routers with the exact same property set as for the router starter. `overrides` can contain a list of properties for the router, which are used in addition to the router starter's properties, or which override the router starter's properties. The property definitions should be separated by white space.

For example, in many cases it is desirable to set the property `Ice.ServerIdleTime` for the router, but not for the router starter. For an idle time of 60 seconds, this can be done by setting `Glacier.Starter.PropertiesOverride=Ice.ServerIdleTime=60`. (Unix only.)

Glacier.Starter.StartupTimeout

Synopsis

`Glacier.Starter.StartupTimeout=num`

Description

num is the number of seconds the Glacier router starter will wait for the router to start up. If this timeout expires, a `Glacier::CannotStartRouterException` is returned to the caller. The default value is 10 seconds. Timeout values smaller than one second are silently changed to 1 second. (Unix only.)

Glacier.Starter.AddUserToAllowCategories

Synopsis

`Glacier.Starter.AddUserToAllowCategories=num`

Description

Control the addition of the user id authenticated by the Glacier router starter to the `Glacier.Router.AllowCategories` property upon router startup:

0	Do not add user id. (default)
1	Add user id.
2	Add user id with prepended underscore.

Glacier.Starter.Trace

Synopsis

`Glacier.Starter.Trace=num`

Description

The router starter trace level: (Unix only.)

0	No router starter trace. (default)
---	------------------------------------

1	Trace router startup exceptions.
2	Also trace each successful router startup.

Glacier.Starter.Certificate.Country

Synopsis

`Glacier.Starter.Certificate.Country=country code`

Description

This property specifies the country code portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. Examples of valid values for this field are "US" for the United States and "CA" for Canada. The default value is "US".

Glacier.Starter.Certificate.StateProvince

Synopsis

`Glacier.Starter.Certificate.StateProvince=state/province code`

Description

This property specifies the state or province code portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. Examples of valid values for this field are "CA" for California and "British Columbia" for British Columbia, Canada. The default value is "DC".

Glacier.Starter.Certificate.Locality

Synopsis

`Glacier.Starter.Certificate.Locality=city or town name`

Description

This property specifies the locality portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. The locality is usually the name of the city or town. The default value is "Washington".

Glacier.Starter.Certificate.Organization**Synopsis**

`Glacier.Starter.Certificate.Organization=organization or company name`

Description

This property specifies the organization portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. The organization is usually the name of the company or organization to which the certificate has been granted. The default value is "Some Company Inc.".

Glacier.Starter.Certificate.OrganizationalUnit**Synopsis**

`Glacier.Starter.Certificate.OrganizationalUnit=department`

Description

This property specifies the organizational unit portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. The organization unit is usually the name of the department within the company or organization structure to which the certificate has been granted. The default value is "Sales".

Glacier.Starter.Certificate.CommonName**Synopsis**

`Glacier.Starter.Certificate.CommonName=contact name`

Description

This property specifies the common name portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. The common name is usually the name of the contact (a person's name) within the company and department to which the certificate has been granted. The default value is "John Doe".

Glacier.Starter.Certificate.BitStrength

Synopsis

`Glacier.Starter.Certificate.BitStrength=number of bits`

Description

This property specifies the bit strength which will be used in the generation of certificates by the Glacier Router Starter for client applications and the Glacier Router itself. This value is the modulus size of the RSA key.

Although modulus size is determined by the needs of the particular application, care should be taken not to specify a size that is too large, as certificate generation is an expensive operation. Sizes less than 512 bits are not supported, and sizes that exceed 2048 bits should be chosen with generation times in mind. The default value is 1024 bits.

Glacier.Starter.Certificate.SecondsValid

Synopsis

`Glacier.Starter.Certificate.SecondsValid=seconds`

Description

This property specifies the number of seconds that certificates generated by the Glacier Router Starter are valid before they will expire. The default value of 1 day (86,400 seconds).

Glacier.Starter.Certificate.IssuedAdjust

Synopsis

`Glacier.Starter.Certificate.IssuedAdjust=(+/-) seconds`

Description

This property adjusts the issued timestamp on dynamically generated certificates for the Glacier Router. The default value is 0, which sets the issued time based on the system time when the certificate is actually created. Positive adjustments will adjust the timestamp into the future, and negative adjustments will cause the timestamp to be in the past by the number of seconds indicated. This adjustment is relative to server time.

C.15 Freeze Properties

Freeze.DbEnv.*env-name*.DbCheckpointPeriod

Synopsis

`Freeze.DbEnv.env-name.DbCheckpointPeriod=num`

Description

Every Berkeley DB environment created by Freeze has an associated thread that checkpoints this environment every *num* seconds. Defaults to 120 seconds.

Freeze.DbEnv.*env-name*.DbHome

Synopsis

`Freeze.DbEnv.env-name.DbHome=db-home`

Description

Defines the home directory of this Freeze database environment. Defaults to *env-name*.

Freeze.DbEnv.*env-name*.DbPrivate

Synopsis

`Freeze.DbEnv.env-name.DbPrivate=num`

Description

If *num* is set to a value larger than zero, Freeze will instruct Berkeley DB to use process private memory instead of shared memory. The default value is 1. Set it to 0 in order to run db_archive (or another Berkeley DB utility) on a running environment.

Freeze.DbEnv.*env-name*.DbRecoverFatal

Synopsis

`Freeze.DbEnv.env-name.DbRecoverFatal=num`

Description

If *num* is set to a value larger than zero, "fatal" recovery will be performed when the environment is opened. The default value is 0.

Freeze.DbEnv.*env-name*.OldLogsAutoDelete

Synopsis

`Freeze.DbEnv.env-name.OldLogsAutoDelete=num`

Description

If *num* is set to a value larger than zero, old transactional logs no longer in use are deleted after each periodic checkpoint (see `Freeze.DbEnv.env-name.DbCheckpointPeriod`). The default value is 1.

Freeze.DbEnv.*env-name*.PeriodicCheckpointMinSize

Synopsis

`Freeze.DbEnv.env-name.PeriodicCheckpointMinSize=num`

Description

num is the minimum size in kbytes for the periodic checkpoint (see `Freeze.DbEnv.env-name.DbCheckpointPeriod`). This value is passed to Berkeley DB's checkpoint function. Defaults to 0 (which means no minimum).

Freeze.Evictor.env-name.filename.MaxTxSize**Synopsis**

`Freeze.Evictor.env-name.filename.MaxTxSize=num`

Description

Freeze uses a background thread to save updates to the database. Transactions are used to save many facets together. *num* defines the maximum number of facets saved per transaction. Defaults to $10 * \text{SaveSizeTrigger}$ (see `Freeze.Evictor.env-name.filename.SaveSizeTrigger`); if this value is negative, the actual value is set to 100.

Freeze.Evictor.env-name.filename.SavePeriod**Synopsis**

`Freeze.Evictor.env-name.filename.SavePeriod=num`

Description

Freeze uses a background thread to save updates to the database. After *num* milliseconds without saving, if there is any facet created, modified or destroyed, this background thread wakes up to save these facets. When *num* is 0, there is no periodic saving. Defaults to 60,000.

Freeze.Evictor.env-name.filename.SaveSizeTrigger**Synopsis**

`Freeze.Evictor.env-name.filename.SaveSizeTrigger=num`

Description

Freeze uses a background thread to save updates to the database. When *num* is 0 or positive, as soon as *num* or more facets have been created, modified or destroyed, this background thread wakes up to save them. When *num* is negative, there is no size trigger. Defaults to 10.

Freeze.Evictor.env-name.filename.StreamTimeout

Synopsis

`Freeze.Evictor.env-name.filename.StreamTimeout=num`

Description

When the saving thread saves an object, it needs to lock this object in order to get a consistent copy of the object's state. If the lock cannot be acquired within *num* seconds, a fatal error is generated. If a fatal error callback was registered by the application, this callback is called; otherwise the program is terminated immediately. When *num* is 0 or negative, there is no timeout. The default value is 0.

Freeze.Trace.DbEnv

Synopsis

`Freeze.Trace.DbEnv=num`

Description

The Freeze database environment activity trace level:

0	No database environment activity trace. (default)
1	Trace database open and close.
2	Also trace checkpoints and the removal of old log files.

Freeze.Trace.Evictor

Synopsis

`Freeze.Trace.Evictor=num`

Description

The Freeze evictor activity trace level:

0	No evictor activity trace. (default)
1	Trace Ice object and facet creation and destruction, facet streaming time, facet saving time, object eviction (every 50 objects) and evictor deactivation.
2	Also trace object lookups, and all object evictions.
3	Also trace object retrieval from the database.

Freeze.Trace.Map

Synopsis

`Freeze.Trace.Map=num`

Description

The Freeze map activity trace level:

0	No map activity trace. (default)
1	Trace database open and close.
2	Also trace iterator and transaction operations, and reference counting of the underlying database.

C.16 Properties

IcePatch.Endpoints

Synopsis

`IcePatch.Endpoints=endpoints`

Description

Specifies the endpoints of the IcePatch server. This property is required by both the IcePatch server and client.

IcePatch.Directory

Synopsis

`IcePatch.Directory=dir`

Description

Sets the IcePatch data directory to *dir*. If not defined, the current working directory is used.

IcePatch.UpdatePeriod

Synopsis

`IcePatch.UpdatePeriod=num`

Description

Sets the IcePatch server's update period to *num* seconds. Updates are executed in a background thread during which the server descends the data directory and performs a number of tasks, including compressing files and computing unique signatures. If not defined, the default value is 60. Automatic updates are disabled if this property is set to zero.

IcePatch.BusyTimeout

Synopsis

`IcePatch.BusyTimeout=num`

Description

Defines the number of seconds *num* the server should wait when attempting to acquire an internal lock on a file or directory. A lock is held whenever a file or directory is being updated, such as when a file is being compressed or a unique signature is being computed. If the server is unable to acquire a lock after the specified timeout period, an `IcePatch::BusyException` is returned. If not specified, the default value is 10.

IcePatch.Trace.Files

Synopsis

`IcePatch.Trace.Files=num`

Description

If *num* is set to a value larger than zero, the server emits diagnostic information to its configured logger. The default value is 0.

IcePatch.RemoveOrphaned

Synopsis

`IcePatch.RemoveOrphaned=num`

Description

If *num* is set to a value larger than zero, the IcePatch client automatically removes files from its local data directory that are no longer present in the server's data directory. The default value is 0.

IcePatch.Thorough

Synopsis

`IcePatch.Thorough=num`

Description

If *num* is set to a value larger than zero, the IcePatch client performs a thorough comparison of each local directory to that of the server. The default value of 0 causes the client to skip a directory if its unique signature matches the server's.

Appendix D

Proxies and Endpoints

D.1 Proxies

Synopsis

```
identity -f facet -t -o -O -d -D -s @ adapter_id :  
endpoints
```

Description

A stringified proxy consists of an identity, proxy options, and an optional object adapter identifier or endpoint list. A proxy containing an identity with no endpoints, or an identity with an object adapter identifier, represents an indirect proxy that will be resolved using the Ice locator (see the `Ice.Default.Locator` property).

Proxy options configure the invocation mode:

-f <i>facet</i>	Select a facet of the Ice object.
-t	Configures the proxy for twoway invocations. (default)

-o	Configures the proxy for oneway invocations.
-O	Configures the proxy for batch oneway invocations.
-d	Configures the proxy for datagram invocations.
-D	Configures the proxy for batch datagram invocations.
-s	Configures the proxy for secure invocations.

The proxy options `-t`, `-o`, `-O`, `-d`, and `-D` are mutually exclusive.

The object identity *identity* is structured as `[category/]name`, where the *category* component and slash separator are optional. If *identity* contains white space or either of the characters `:` or `@`, it must be enclosed in single or double quotes. The *category* and *name* components are UTF8 strings that use the encoding described below. Any occurrence of a slash `(/)` in *category* or *name* must be escaped with a backslash (i.e., `\/`).

The *facet* argument of the `-f` option represents a facet path comprising one or more facets separated by a slash `(/)`. If *facet* contains white space, it must be enclosed in single or double quotes. Each component of the facet path is a UTF8 string that uses the encoding described below. Any occurrence of a slash `(/)` in a facet path component must be escaped with a backslash (i.e., `\/`).

The object adapter identifier *adapter_id* is a UTF8 string that uses the encoding described below. If *adapter_id* contains white space, it must be enclosed in single or double quotes.

UTF8 strings are encoded using ASCII characters for the ordinal range 32-126 (inclusive). Characters outside this range must be encoded using escape sequences (`\b`, `\f`, `\n`, `\r`, `\t`) or octal notation (e.g., `\007`). Quotation marks used to enclose a string can be escaped using a backslash, as can the backslash itself (`\\"`).

If endpoints are specified, they must be separated with a colon `(:)` and formatted as described in Endpoints. The order of endpoints in the stringified proxy is not necessarily the order in which connections are attempted during binding: when a stringified proxy is converted into a proxy instance, the endpoint list is randomized as a form of load balancing.

If the `-s` option is specified, only those endpoints that support secure invocations are considered during binding. If no valid endpoints are found, the application receives `Ice::NoEndpointException`.

Otherwise, if the `-s` option is not specified, the endpoint list is ordered so that non-secure endpoints have priority over secure endpoints during binding. In other

words, connections are attempted on all non-secure endpoints before any secure endpoints are attempted.

If an unknown option is specified, or the stringified proxy is malformed, the application receives `Ice::ProxyParseException`. If an endpoint is malformed, the application receives `Ice::EndpointParseException`.

D.2 Endpoints

Synopsis

```
endpoint : endpoint
```

Description

An endpoint list comprises one or more endpoints separated by a colon (:). An endpoint has the following format: ***protocol option*** The supported protocols are `tcp`, `udp`, `ssl`, and `default`. If `default` is used, it is replaced by the value of the `Ice.Default.Protocol` property. If an endpoint is malformed, or an unknown protocol is specified, the application receives `Ice::EndpointParseException`.

The `ssl` protocol is only available if the IceSSL plug-in is installed.

The protocols and their supported options are described in the sections that follow.

TCP Endpoint

Synopsis

```
tcp -h host -p port -t timeout -z
```

Description

A `tcp` endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
<code>-h host</code>	Specifies the hostname or IP address of the endpoint. If not specified, the value of <code>Ice.Default.Host</code> is used instead.	Determines the hostname or IP address to which a connection attempt is made.	Determines the network interface on which the object adapter listens for connections, as well as the hostname that is advertised in proxies created by the adapter.
<code>-p port</code>	Specifies the port number of the endpoint.	Determines the port to which a connection attempt is made. (required)	The port will be selected by the operating system if this option is not specified or <code>port</code> is zero.
<code>-t timeout</code>	Specifies the endpoint timeout in milliseconds.	If <code>timeout</code> is greater than zero, it sets a maximum delay for binding and proxy invocations. If a timeout occurs, the application receives <code>Ice::TimeoutException</code> .	Determines the default timeout that is advertised in proxies created by the object adapter.

Option	Description	Client Semantics	Server Semantics
-z	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compressed responses are sent, as well as whether compression is advertised in proxies created by the adapter.

UDP Endpoint

Synopsis

```
udp -v major.minor -e major.minor -h host -p port -c -z
```

Description

A `udp` endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
<code>-v major. minor</code>	Specifies the protocol major and highest minor version number to be used for this endpoint. If not specified, the protocol major version and highest supported minor version of the client-side Ice run time is used.	Determines the protocol major version and highest minor version used by the client side when sending messages to this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.	Determines the protocol major version and highest minor version advertised by the server side for this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.

Option	Description	Client Semantics	Server Semantics
<code>-e <i>major. minor</i></code>	Specifies the encoding major and highest minor version number to be used for this endpoint. If not specified, the encoding major version and highest supported minor version of the client-side Ice run time is used.	Determines the encoding major version and highest minor version used by the client side when sending messages to this endpoint. The encoding major version number must match the encoding major version number of the server; the encoding minor version number must not be higher than the highest minor version number supported by the server.	Determines the encoding version and highest minor version advertised by the server side for this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.

Option	Description	Client Semantics	Server Semantics
<code>-h host</code>	Specifies the hostname or IP address of the endpoint. If not specified, the value of <code>Ice.Default.Host</code> is used instead.	Determines the hostname or IP address to which datagrams are sent.	Determines the network interface on which the object adapter listens for datagrams, as well as the hostname that is advertised in proxies created by the adapter.
<code>-p port</code>	Specifies the port number of the endpoint.	Determines the port to which datagrams are sent. (required)	The port will be selected by the operating system if this option is not specified or <i>port</i> is zero.
<code>-c</code>	Specifies that a connected UDP socket should be used.	None.	Causes the server to connect to the socket of the first peer that sends a datagram to this endpoint.
<code>-z</code>	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compression is advertised in proxies created by the adapter.

SSL Endpoint

Synopsis

```
ssl -h host -p port -t timeout -z
```

Description

An `ssl` endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
<code>-h <i>host</i></code>	Specifies the hostname or IP address of the endpoint. If not specified, the value of <code>Ice.Default.Host</code> is used instead.	Determines the hostname or IP address to which a connection attempt is made.	Determines the network interface on which the object adapter listens for connections, as well as the hostname that is advertised in proxies created by the adapter.
<code>-p <i>port</i></code>	Specifies the port number of the endpoint.	Determines the port to which a connection attempt is made. (required)	The port will be selected by the operating system if this option is not specified or <i>port</i> is zero.

Option	Description	Client Semantics	Server Semantics
<code>-t timeout</code>	Specifies the endpoint timeout in milliseconds.	If <i>timeout</i> is greater than zero, it sets a maximum delay for binding and proxy invocations. If a timeout occurs, the application receives <code>Ice::TimeoutException</code> .	Determines the default timeout that is advertised in proxies created by the object adapter.
<code>-z</code>	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compressed responses are sent, as well as whether compression is advertised in proxies created by the adapter.

Bibliography

References

1. Booch, G., et al. 1998. *Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.
2. Gamma, E., et al. 1994. *Design Patterns*. Reading, MA: Addison-Wesley.
3. Grimes, R. 1998. *Professional DCOM Programming*. Chicago, IL: Wrox Press.
4. Henning, M., and S. Vinoski. 1999. *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.
5. Housley, R., and T. Polk. 2001. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*. Hoboken, NJ: Wiley.
6. Institute of Electrical and Electronics Engineers. 1985. *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*. Piscataway, NJ: Institut of Electrical and Electronic Engineers.
7. Jain, P., et al. 1997. “Dynamically Configuring Communication Services with the Service Configurator Pattern.” *C++ Report* 9 (6).
8. Kleiman, S., et al. 1995. *Programming With Threads*. Englewood, NJ: Prentice Hall.
9. Lakos, J. 1996. *Large-Scale C++ Software Design*. Reading, MA: Addison-Wesley.

10. McKusick, M. K., et al. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Reading, MA: Addison-Wesley.
11. Microsoft. 2002. *.NET Infrastructure & Services*.
<http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/net/Default.asp>. Bellevue, WA: Microsoft.
12. Mitchell, J. G., et al. 1979. *Mesa Language Manual*. CSL-793. Palo Alto, CA: Xerox PARC.
13. Object Management Group. 2001. *Unified Modeling Language Specification*.
<http://www.omg.org/technology/documents/formal/uml.htm>. Framingham, MA: Object Management Group.
14. The Open Group. 1997. *DCE 1.1: Remote Procedure Call*. Technical Standard C706. <http://www.opengroup.org/publications/catalog/c706.htm>. Cambridge, MA: The Open Group.
15. Prescod, P. 2002. *Some Thoughts About SOAP Versus REST on Security*.
<http://www.prescod.net/rest/security.html>.
16. Red Hat, Inc. 2003. *The bzip2 and libbzip2 Home Page*.
<http://sources.redhat.com/bzip2>. Raleigh, NC: Red Hat, Inc.
17. Schmidt, D. C. et al. 2000. “Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching”. In *Proceedings of the 7th Pattern Languages of Programs Conference*, WUCS-00-29, Seattle, WA: University of Washington. <http://www.cs.wustl.edu/~schmidt/PDF/lf.pdf>.
18. Sleepycat Software, Inc. 2003. *Berkeley DB Technical Articles*.
<http://www.sleepycat.com/company/technical.shtml>. Lincoln, MA: Sleepycat Software, Inc.
19. Sutter, H. 1999. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Reading, MA: Addison-Wesley.
20. Sutter, H. 2002. “A Pragmatic Look at Exception Specifications.” *C/C++ Users Journal* 20 (7): 59–64. <http://www.gotw.ca/publications/mill22.htm>.
21. Unicode Consortium, ed. 2000. *The Unicode Standard, Version 3.0*. Reading, MA: Addison-Wesley. <http://www.unicode.org/unicode/uni2book/u2.html>.
22. Viega, J., et al. 2002. *Network Security with OpenSSL*. Sebastopol, CA: O'Reilly.
23. World Wide Web Consortium. 2002. *SOAP Version 1.2 Specification*.
<http://www.w3.org/2000/xp/Group/#soap12>. Boston, MA: World Wide Web Consortium.

24. World Wide Web Consortium. 2002. *Web Services Activity*.
<http://www.w3.org/2002/ws>. Boston, MA: World Wide Web Consortium.

