GEMSTONE®

GEMFIRE®
ENTERPRISE

Native Client Guide

Version 3.5

September 2010

GemStone Systems, Inc. 503•533•3000 www.gemstone.com

GemStone Systems, Inc.
1260 NW Waterhouse Avenue, Suite 200
Beaverton, OR 97006

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

About This Manual

This guide describes how to use the GemStone® GemFire Enterprise® native client interface.

How This Manual Is Organized

This manual contains the following chapters:

Typographical Conventions

This document uses the following typographical conventions:

- Methods, types, file names and paths, code listings, and prompts are shown in `Courier New` typeface. For example:

  ```
  put
  ```

- Parameters and variables are shown in *italic* font. For example,

  `gfConnect(`*`sysDir, connectionName, writeProtectAllowed`*`)`

- In examples showing both user input and system output, the lines you type are distinguished from system output by **`boldface`** type:

  `prompt>` **`gemfire`**

- If you are viewing this document online, the page, section, and chapter references are hyperlinks to the places they refer to, like this reference to *System Requirements* on page 25 and this reference to Chapter 1, *Introducing the GemFire Enterprise Native Client*, on page 23. Blue text denotes a hyperlink.

Other Useful Documents

The GemFire Enterprise native client C++ and .NET API reference pages can be accessed through the file `native.html` in the `docs` directory. The `docs` directory is located at *`productDir`*`/docs`, where *`productDir`* is the path to the native client product directory.

This manual, the *GemFire Enterprise Native Client Guide*, describes the installation, configuration and administration requirements and procedures for the native client.

Technical Support

GemStone provides several sources for product information and support. This manual, the *GemFire Enterprise Native Client Guide,* and the API reference pages provide extensive documentation and should always be your first source of information for the native client. GemStone Technical Support engineers will refer you to these documents when applicable. However, you may need to contact Technical Support for the following reasons:

- Your technical question is not answered in the documentation
- You receive an error message that directs you to contact GemStone Technical Support
- You want to report a bug
- You want to submit a feature request

Questions concerning product availability, pricing, license keyfiles, or future features should be directed to your GemStone account manager.

Contacting Technical Support

When contacting GemStone Technical Support, be prepared to provide the following information:

- Your name, company name, and GemFire license number
- The GemFire product and version you are using
- The hardware platform and operating system you are using
- A description of the problem or request
- Exact error messages received, if any
- Any artifacts

Your GemStone support agreement may identify specific individuals who are responsible for submitting all support requests to GemStone. If so, please submit your information through those individuals. All responses will be sent to authorized contacts only.

For non-emergency requests, you should contact Technical Support by web form or e-mail. You will receive confirmation of your request, and a request assignment number for tracking. Replies will be sent by e-mail whenever possible, regardless of how they were received.

GemStone Support Website: http://techsupport.gemstone.com

This is the preferred method of contact. The Help Requests link is at the top right corner of the home page. Use the form to submit help requests. The form requires an account, but registration is free of charge. To get an account, complete the New User Registration form found in the same location. You will be able to access the site as soon as you submit the web form.

E-mail: techsupport@gemstone.com

Please do not send files larger than 100K to this e-mail address (core dumps, for example). A special address for large files will be provided as appropriate.

Telephone: (800) 243-4772 or (503) 533-3503

We recommend that you call only for more serious requests that require immediate evaluation, such as a production system that is non-operational.

Emergency requests are handled by the first available engineer. If you are reporting an emergency and you receive a recorded message, do not use the voicemail option. Transfer your call to the operator, who will take a message and immediately contact an engineer.

Non-emergency requests received by telephone are placed in the normal support queue for evaluation and response.

24x7 Emergency Technical Support

GemStone offers 24x7 emergency technical support at an additional charge. This support entitles customers to contact GemStone 24 hours a day, 7 days a week, 365 days a year, if they encounter problems that cause their production application to go down or that have the potential to bring their production application down. Contact your GemStone account manager for more details.

GemStone Support Web Site

The GemStone support web site, http://techsupport.gemstone.com, provides a variety of information to help you use GemStone products. Use of this site requires an account, but registration is free of charge. To get an account, just complete the Registration Form found in the same location. You will be able to access the site as soon as you submit the web form.

The following types of information are provided at this web site:

Help Request is an online form that allows designated technical support contacts to submit requests for information or assistance via e-mail to GemStone Technical Support.

Technotes provide answers to questions submitted by GemStone customers. They may contain coding examples, links to other sources of information, or downloadable code.

Bugnotes identify performance issues or error conditions that you may encounter when using a GemStone product. A bugnote describes the cause of the condition and, when possible, provides an alternative means of accomplishing the task. In addition, bugnotes identify whether a fix is available, either by upgrading to another version of the product or by applying a patch. Bugnotes are updated regularly.

Patches provide code fixes and enhancements that have been developed after product release. A patch generally addresses a specific group of behavior or performance issues. Most patches listed on the GemStone web site are available for direct downloading.

Tips and Examples provide information and instructions for topics that usually relate to more effective or efficient use of GemStone products. Some tips may contain code that can be downloaded for use at your site.

Release Notes and Install Guides for product software are provided in PDF and HTML format.

Documentation for GemStone GemFire is provided in PDF and HTML format.

Community Links provide customer forums for discussion of GemStone product issues.

Technical information on the GemStone web site is reviewed and updated regularly. We recommend that you check this site on a regular basis to obtain the latest technical information for GemStone products. We also welcome suggestions and ideas for improving and expanding the site to better serve you.

Training and Consulting

Consulting and training for all GemStone products is available through GemStone’s Professional Services organization.

- Training courses are offered periodically at GemStone’s offices in Beaverton, Oregon, or you can arrange for onsite training at your desired location.
- Customized consulting services can help you make the best use of GemStone products in your business environment.

Contact your GemStone account representative for more details or to obtain consulting services.

Chapter 1

Introducing the GemFire Enterprise Native Client

The GemFire Enterprise native client provides access for C++ and Microsoft® .NET™ clients to the GemFire Enterprise distributed system. The native client is written entirely in C++, so its initialization process does not involve the creation of a Java virtual machine. The .NET native client provides native operations for the .NET Framework application developer writing in .NET languages that needs to access the GemFire cache server.

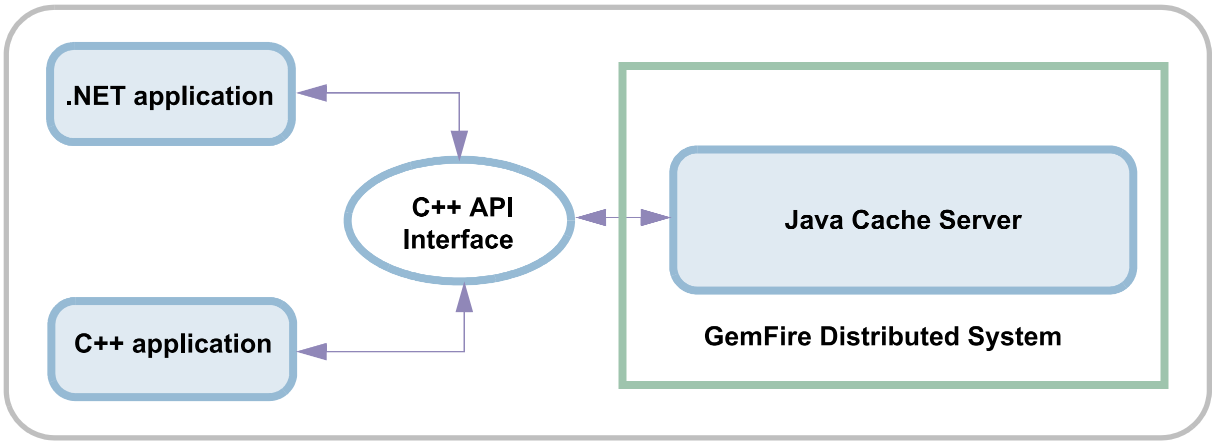
GemFire Enterprise native clients in C++, Java, and .NET languages communicate only with the cache server and do not communicate with each other. The native clients interface with the server at the sockets level and implement the same wire protocol to the server. These capabilities produce extremely high performance and system scalability.

C++ and .NET native clients provide access to the full region API, including support for application plug-ins, managed connectivity, highly available data, and reliable failover to a specified server list. All of this is transparent to the end user.

The native client delivers the full set of capabilities supplied by Java clients communicating with the GemFire Enterprise cache server. You can configure GemFire Enterprise native clients to cache data locally, or they can act in a cacheless mode where they retrieve data from a cache server and directly pass it to other system members without incurring the caching overhead. They can be configured as read only caches, or be configured to receive notifications from the server whenever a key of interest to the client changes on the server.

This is a conceptual overview of how .NET and C++ applications access the cache server.

Figure 1.1 GemFire Enterprise Native Client Overview



In this chapter:

1.1 System Requirements

The GemFire Enterprise native client operates on platforms running Microsoft Windows, Linux (Intel), and Sun Solaris, as listed in the chart below.

Operating System	Minimum Platform Requirement	RAM	Swap Space	Disk Space Required
Windows XP Professional Windows 2003 Server	Intel® Pentium® IV 2 GHz (or equivalent)	2 GB	256 MB of virtual memory	80 MB
Linux	Intel® Pentium® IV 2 GHz (or equivalent)	2 GB	256 MB of virtual memory	70 MB
Solaris	Sun SPARC Blade 1000 with V9 instruction set	2 GB	256 MB of virtual memory	70 MB

Windows

- The native client is built and tested on Windows XP Professional, Service Pack 2.
- The native client is not supported on Windows NT 4.0.
- The native client is supported with Microsoft .NET Framework 2.0, 3.0, and 3.5.
- Microsoft .NET Framework Version 2.0 must be installed to support C++/CLI (Common Language Infrastructure) for the native client.

  You can download the .NET Framework Version 2.0 Redistributable Package (x86 for 32bit or x64 for 64bit) from http://www.microsoft.com/downloads. If it isn't listed on the Download Center page, use the Search tool to search for '.NET Framework Version 2.0'.

Linux

The GemFire Enterprise native client is built on Red Hat Enterprise ES 3, kernel version 2.4.21-47.EL.

The native client is tested on the following Linux versions:

- SLES 9 SP3, kernel version 2.6.5-7.244-smp
- SLES 10 SP1, kernel version 2.6.16.27-0.9-smp
- Red Hat Enterprise ES 4, kernel version 2.6.9-27.ELsmp

  Update 3 (Nahant) or later is recommended.
- Red Hat Enterprise 5 release 5 (Tikanga), kernel version 2.6.18-8.EL5
- Red Hat Enterprise WS 3 and WS 4, kernel version 2.4.21-40.ELsmp

If you are not sure of the kernel version on your system, use this command to list it:

```
prompt> uname -r
```

The following table lists the RPM package dependencies for several Linux distributions.The i386 or i686 after the package name indicates that you must install the package for that particular architecture regardless of the native operating system architecture. All of the packages listed are available with the default media for each distribution.

Table 1.1 GemFire Dependencies on Linux RPM Packages

Linux Version	glibc	libgcc
Red Hat Enterprise Linux ES release 3 (i686)	`glibc`	`libgcc`
Red Hat Enterprise Linux ES release 3 (x86_64)	`glibc (i686)`	`libgcc (i686)`
Red Hat Enterprise Linux ES release 4 (i686)	`glibc`	`libgcc`
Red Hat Enterprise Linux ES release 4 (x86_64)	`glibc (i686)`	`libgcc (i386)`
Red Hat Enterprise Linux Server release 5 (i686)	`glibc`	`libgcc`
Red Hat Enterprise Linux Server release 5 (x86_64)	`glibc (i686)`	`libgcc (i386)`
SUSE LINUX Enterprise Server 9 (i586)	`glibc`	`libgcc`
SUSE LINUX Enterprise Server 9 (x86_64)	`glibc-32bit`	`libgcc`
SUSE Linux Enterprise Server 10 (x86_64)	`glibc-32bit`	`libgcc`

For versions of Linux not listed in the table, you can verify that you meet the native client dependencies at the library level by using the `ldd` tool and entering this command:

```
prompt> ldd $GFCPP/lib/libgfcppcache.so
```

The following libraries are external dependencies of the native library, `libgfcppcache.so`. Verify that the `ldd` tool output includes all of these:

```
libdl.so.2
libm.so.6
libpthread.so.0
libc.so.6
libz.so.1
```

For details on the `ldd` tool, see its Linux online man page.

Solaris

The native client is supported on the following Solaris versions:

- Solaris 9 kernel update 118558-38
- Solaris 10 kernel update 118833-24

1.2 Installing and Uninstalling the Native Client

This section provides installation instructions and additional environment setup details for the GemFire Enterprise native client. Instructions are also supplied for uninstalling the product.

After the native client is installed, a license file must be supplied so the features you require are enabled. For more information about native client licensing, see Licensing (page 30).

Installing on Linux and Solaris

For Linux and Solaris, the native client is installed by extracting the contents of a ZIP file.

When the installation is complete, you will need to add these environment settings:

- Set the `GFCPP` environment variable to *`productDir`*, where *`productDir`* represents the `NativeClient_xxxx` directory (`xxxx` is the four-digit product version).
- Add `$GFCPP/bin` to the `PATH`.

Linux

The Linux installer is `NativeClient_xxxx_Lin.zip`. The default installation path is `NativeClient_xxxx`, where `xxxx` represents the four-digit product version.

Solaris

The Solaris installer is `NativeClient_xxxx_Sol.zip`. The default installation path is `NativeClient_xxxx`, where `xxxx` represents the four-digit product version.

Installing on Windows

The native client can be installed on Windows by using the `NativeClient_xxxx.msi` Windows installer, where `xxxx` represents the four-digit version of the product. The installer requires `msiexec` version 3.0 or higher. Version 3.0 is standard on Windows XP SP2.

Double-click the MSI file to start the installation. You can accept the default installation path, or install to a different location. The product is installed in `NativeClient_xxxx`, where `xxxx` represents the four-digit product version.

You must be logged in with administrative privileges or the MSI installation will fail.

To complete the installation, the MSI installer automatically configures these native client system environment settings:

- Sets the `GFCPP` environment variable to *`productDir`*, where *`productDir`* represents the `NativeClient_xxxx` directory (`xxxx` is the four-digit product version).
- Adds `%GFCPP%\bin` to the Windows `PATH`.

Installing Using the MSI Command-Line

In addition to the standard Windows installer interface, the `NativeClient_xxxx.msi` file provides a set of command-line installer options. This is an example of the installer command-line syntax:

```
msiexec /q /i NativeClient_xxxx.msi DEFAULT_INSTALLDIR=<path>
```

The following table lists common `msiexec` command-line options for use with `NativeClient_xxxx.msi`, along with an explanation of their usage. For a list of all command-line options, enter `msiexec/?`.

Table 1.2 MSI Command-Line Options

Option	Explanation
`/q`	Creates a quiet installation with no interface or prompts.
`/i`	Indicates that the product is to be installed or configured.
`DEFAULT_INSTALLDIR=<path>`	Specifies the destination directory, if different from the default.
`/x`	Indicates a product uninstall procedure.

Repairing a Windows Installation

If problems occur with your Windows native client installation you can initiate a repair operation to restore any missing elements or registry settings. Follow these steps:

1. Double-click the MSI file, then click Next in the Setup Wizard screen.
2. In the following screen, click the Repair button.
3. In the next screen, click the Repair button.

Uninstalling the Native Client

The native client can be uninstalled in different ways, depending on your operating system.

Linux and Solaris

On Linux and Solaris, the native client can be uninstalled by deleting the `NativeClient_xxxx` product directory and all of its subdirectories.

Windows

The native client can be uninstalled on Windows using either the MSI installer graphical interface, its command-line interface, or through the Windows Control Panel.

Using the MSI Installer Graphical Interface

1. Double-click the MSI file, then click Next in the Setup Wizard screen.
2. In the following screen, click the Remove button.
3. In the next screen, click the Remove button and then click Finish.

Using the MSI Installer Command-Line

This is the MSI command-line syntax for a quiet uninstall:

```
msiexec /q /x NativeClient_xxxx.msi DEFAULT_INSTALLDIR=<path>
```

As described in Table 1.2, The `/x` option uninstalls the product.

Using the Control Panel

1. From the Control Panel, open Add or Remove Programs.
2. Select `Gemfire Native Client x.x.x.x`.
3. Click the Remove button.

1.3 Licensing

The native client requires a license for full functionality with GemFire. You must specifically request a native client license when you contact GemStone Technical Support.

An application can reference the license file by using a `gfcpp.properties` configuration file that points to where the license file is located. For example, this setting entered in `gfcpp.properties` points to the license file that is in the directory where the `gfcpp.properties` file is located:

```
license-file=gfCppLicense.zip
```

The license generates a log file for examination if any licensing errors occur. The log file may help you and the GemStone Technical Support team resolve the licensing problem.

1.4 Running Native Client Applications

This section provides information about setting up the environment for the native client on multiple platforms, and for compiling and running client programs.

In this section:

- Developing C++ Programs on Linux (page 31)
- Developing C++ Programs on Windows (page 33)
- Developing C++ Programs on Solaris (page 35)

Developing C++ Programs on Linux

Setting the GemFire Environment Variables

Before you begin, set the native client environment variables on each Linux host. For each case, *productDir* is the path to the native client product directory.

For Bourne and Korn shells (sh, ksh, bash)

```
GFCPP=productDir; export GFCPP
PATH=$GFCPP/bin:$PATH;export PATH
LD_LIBRARY_PATH=$GFCPP/lib:$LD_LIBRARY_PATH;export LD_LIBRARY_PATH
```

Compiling and Linking

On Linux, the `g++` compiler is supported. To build and link a C++ client to GemFire Enterprise on Linux, the compilation command line must include the arguments listed in Table 1.3.

Table 1.3 Compiler Arguments (Linux)

Argument	Explanation
`-D_REENTRANT`	Required to compile Linux programs in a thread-safe way.
`-m32`	Enables 32-bit compilation.
`-m64`	Enables 64-bit compilation.
`-I$GFCPP/include`	Specifies the native client `include` directory.

Dynamically Linking

The following table lists the linker switches that must be present on the command line when dynamically linking to the GemFire library.

Table 1.4 Linker Switches (Dynamically Linking on Linux)

Argument	Explanation
`-rpath $GFCPP/lib`	Tells the linker to look in `$GFCPP/lib` for libraries on which the GemFire library depends
`-L$GFCPP/lib`	Tells the linker where to find the named libraries.
`-lgfcppcache`	Links the GemFire C++ cache library to the compiled executable.

The following example compiles and links the `$GFCPP/examples/cacheRunner/CacheRunner.cpp` client to `a.out`.

Example 1.1 Compiling and Dynamically Linking on Linux for 32-bit

```
g++ -D_REENTRANT -m32 -I$GFCPP/include -rpath \
$GFCPP/lib -L$GFCPP/lib \
$GFCPP/examples/cacheRunner/CacheRunner.cpp -o cacherunner -lgfcppcache
```

Example 1.2 Compiling and Dynamically Linking on Linux for 64-bit

```
g++ -D_REENTRANT -m64 -I$GFCPP/include -rpath \
$GFCPP/lib -L$GFCPP/lib \
$GFCPP/examples/cacheRunner/CacheRunner.cpp -o cacherunner -lgfcppcache
```

Loading the GemFire Library

When the C++ application is dynamically linked to the GemFire library, the library must be dynamically loadable. To make the GemFire library available for load, add the path *`GemFireDir`*`/lib` to the `LD_LIBRARY_PATH` environment variable, where *`GemFireDir`* is the path to the GemFire product directory.

Developing C++ Programs on Windows

GemFire uses the Visual Studio 2005 Service Pack 1 compiler for C++ programs on Windows, which invokes Microsoft® `cl.exe` from the command line at compile time. Visual Studio 2005 SP1 is the recommended compiler. If you are using any other compiler, contact GemStone Technical Support for assistance.

GemFire Environment Variables

When you install on Windows, the installer performs these tasks:

- Sets the `GFCPP` environment variable to *`productDir`*, where *`productDir`* is the path to the native client product directory.
- Adds the `%GFCPP%\bin` executable directory to the Windows `PATH`.

Choosing a 32-bit or 64-bit command-line prompt

For 32bit: Start->Programs->Microsoft Visual Studio 2005->Visual Studio Tools->Visual Studio 2005 Command Prompt

For 64bit: Start->Programs->Microsoft Visual Studio 2005->Visual Studio Tools->Visual Studio 2005 x64 Win64 Command Prompt

To build using the Microsoft Visual Studio Interface, from the Solutions Platform, choose Win32 from the for 32bit build or x64 for the 64bit build.

Compiling and Linking

The following table lists the compiler and linker switches that must be present on the `cl.exe` command line.

If you want to use the Visual Studio user interface instead of invoking `cl.exe` *from the command line, be sure to supply these parameters.*

Table 1.5 Compiler and Linker Switches for Windows

Argument	Explanation
`/MD`	Memory model.
`/EHsc`	Catches C++ exceptions only and tells the compiler to assume that *extern* C functions never throw a C++ exception.
`/GR`	Runtime type information.
`-I%GFCPP%\include`	Specifies the GemFire `include` directory.
`%GFCPP%\lib\gfcppcache.lib`	Specifies the library file for the shared library.
`/D_CRT_SECURE_NO_DEPRECATE`	Suppresses warnings. Required for Visual Studio 2005.
`/D_CRT_NON_CONFORMING_SWPRINTFS`	Suppresses warnings. Required for Visual Studio 2005.

The next example compiles and links the `$GFCPP\examples\cacheRunner\CacheRunner.cpp` client to `CacheRunner.exe`.

Example 1.3 Compiling and Linking on Windows

```
cl /MD /GX /GR /DWIN32 /D_WIN32_WINNT=0x0500 -I%GFCPP%\include \
%GFCPP%\examples\cacheRunner\CacheRunner.cpp %GFCPP%\lib\gfcppcache.lib
```

Loading the GemFire Library

Because GemFire does not provide a library that can be linked statically into an application, on Windows you must dynamically link to the GemFire library.

To make the GemFire library available for loading, verify that the directory `*GemFireDir*/bin` is included in the `PATH` environment variable, where *`GemFireDir`* is the path to the GemFire product directory.

Developing C++ Programs on Solaris

Setting the GemFire Environment Variables

Before you begin, set the native client environment variables on each Solaris host. For each case, `productDir` is the path to the native client product directory.

For Bourne and Korn shells (sh, ksh, bash)

```
GFCPP=productDir; export GFCPP
PATH=$GFCPP/bin:$PATH;export PATH
LD_LIBRARY_PATH=$GFCPP/lib:$LD_LIBRARY_PATH;export LD_LIBRARY_PATH
```

Compiling and Linking

Version 5.9 of the `SUNpro` compiler is supported on Solaris. The linker switches vary according to whether you are statically linking to the GemFire library.

To build and link a C++ client to GemFire Enterprise on Solaris, the compilation command line must include the appropriate arguments from this table.

Table 1.6 Compiler Arguments on Solaris

Argument	Explanation
`-D_REENTRANT`	Required to compile Solaris programs in a thread-safe way.
`-xarch=v8plus`	Enables 32-bit compilation.
`-xarch=v9`	Enables 64-bit compilation.
`-ldl; -lpthread; -lc; -lm; -lsocket; -lrt; -lnsl; -ldemangle; -lkstat; -lz`	Additional libraries.
`-library=stlport4`	Solaris library compilation.
`-I$GFCPP/include`	Specifies the GemFire include directory.

Loading the GemFire Library

When a C++ application is not statically linked to the GemFire library, the library must be dynamically loadable. To make the GemFire library available for load, add the path *`GemFireDir`*`/lib` to `LD_LIBRARY_PATH`, where *`GemFireDir`* is the path to the GemFire product directory.

1.5 Running the Product Examples and QuickStart Guide

Product Examples

The product examples demonstrate the capabilities of the GemFire Enterprise native client, and provide source code so you can examine how each example is designed. Both C++ and C# examples are available to see how the native client performs as either a C++ or C# client. Each example has a `README.HTML` that provides setup and operating instructions.

The examples are located in the subdirectories of the *`productDir`*`/examples` directory, where *`productDir`* is the path to the native client product directory.

The C# examples have a `.config` *file that should not be modified or deleted. The file provides references to support files for running the example.*

Configuring Output

You can control the output of the product examples through an optional system-level configuration file. Follow these steps:

1. If it is not present, copy *`productDir`*`/defaultSystem/gfcpp.properties` to the example directory.
2. To decrease the volume of messages, open the copy of `gfcpp.properties` in the example directory, uncomment the `log-level` line, then change its setting to `error`:
   ```
   log-level=error
   ```
3. To send the messages to a log file instead of `stdout`, uncomment the `log-file` line and specify a name for the log file:
   ```
   log-file=gemfire_cpp.log
   ```

QuickStart Guide

The QuickStart Guide for the native client consists of a set of compact programming samples that demonstrate both C++ and C# client operations. Run the QuickStart Guide to rapidly become familiar with native client functionality. The QuickStart has a `README.HTML` file in *`productDir`*`/quickstart` that describes each programming sample, explains initial environment setup, and provides instructions for running the QuickStart.

Chapter 2

The Native Client Cache

The GemFire Enterprise native client cache provides a framework for native clients to store, manage, and distribute application data. The native client cache provides the following features:

- Local and distributed data caching for fast access.
- Data distribution between applications on the same or different platforms.
- Local and remote data loading through application plug-ins.
- Application plug-ins for synchronous and asynchronous handling of data events.
- Automated and application-specific data eviction for freeing up space in the cache, including optional overflow to disk.
- System message logging, and statistics gathering and archiving.

In this chapter:

2.1 Using Thread Safety in Cache Management

When performing structural changes on your cache, such as creating or closing a `Cache`, `Pool`, or `Region`, synchronize your operations or do them in a single thread.

Other non-structural operations, like Region gets, puts, and queries, are thread safe and you may perform them in a multithreaded way. There are caveats to this, for example, when two threads update the same key simultaneously, there is no way to determine which thread's operation will prevail.

You may need to protect cached objects from concurrent usage and modification. The native client does not guard cached objects themselves from concurrent access.

Always catch and handle exceptions that may be thrown, for problems like trying to create a `Pool` with the same name more than once.

2.2 Caches

The cache is the entry point to native client data caching in GemFire. Through the cache, native clients gain access to the GemFire caching framework for data loading, distribution, and maintenance.

The cache is composed of a number of data regions, each of which can contain any number of entries. Region entries hold the cached data. Every entry has a key that uniquely identifies it within the region and a value where the data object is stored.

Regions are created from the `Cache` instance. Rgions provide the entry point to the interfaces for instances of `Region` and `RegionEntry`.

This section describes cache management. Subsequent sections cover region and entry management.

The Caching APIs

RegionService

`RegionService` provides:

- Access to existing cache regions.
- Access to the standard query service for the cache, which sends queries to the servers. See Chapter 9, *Remote Querying*, on page 179 and Chapter 10, *Continuous Querying*, on page 221.

`RegionService` is inherited by `Cache`.

You do not use instances of `RegionService` except for secure client applications with many users. See *Creating Multiple Secure User Connections with RegionService* on page 168.

Cache

Use the `Cache` to manage your client caches. You have one `Cache` per client.

The `Cache` inherits `RegionService` and adds management of these client caching features:

- Region creation.
- Subscription keepalive management for durable clients.
- Access to the underlying distributed system.
- `RegionService` creation for secure access by multiple users.

Cache Ownership

The distributed system defines how native client and cache server processes find each other. The distributed system keeps track of its membership list and makes its members aware of the identities of the other members in the distributed system.

A cache for a native client is referred to as its local cache. All other caches in the distributed system are considered remote caches to the application. Every cache server and application process has its own cache. The term distributed cache is used to describe the union of all caches in a GemFire distributed system.

Creating and Accessing a Cache

You provide this information in input to the cache creation:

- The connection name, used in logging to identify both the distributed system connection and the cache instance. If you do not specify a connection name, a unique (but non-descriptive) default name is assigned.
- The `cache.xml` to initialize the cache (if the initialization is not done programmatically). To modify the cache structure, you just edit `cache.xml` in your preferred text editor. No changes to the application code are required. If you do not specify a cache initialization file, you need to initialize the cache programmatically.

  The `cache.xml` file contains XML declarations for cache, region, and region entry configuration.

  This XML declares server connection pools and regions:

  ```
  <cache>
     <region name="clientRegion1" refid="PROXY">
        <region-attributes pool-name="serverPool1"/>
     </region>
     <region name="clientRegion2" refid="PROXY">
        <region-attributes pool-name="serverPool2"/>
     </region>
     <region name="localRegion3" refid="LOCAL"/>
     <pool name="serverPool1">
        <locator host="host1" port="40404"/>
     </pool>
     <pool name="serverPool2">
        <locator host="host2" port="40404"/>
     </pool>
  </cache>
  ```

  When you use the regions, the client regions connect to the servers through the pools named in their configurations.

  This file can have any name, but is generally referred to as `cache.xml`.

  For a list of the parameters in the `cache.xml` file, including the DTD, see Chapter 3, *Cache Initialization File*, on page 75.

To create your cache, call the `CacheFactory` `create` function. The `cache` object it returns gives access to the native client caching API. Example:

```
CacheFactoryPtr cacheFactory = CacheFactory::createCacheFactory();
CachePtr cachePtr = cacheFactory->create();
```

Cache Functionality

The `Cache` instance allows your process to do the following:

- Set general parameters for communication between a cache and other caches in the distributed system.
- Create and access any region in the cache. This provides the entry point to region and entry management in the cache.

Closing the Cache

Use the `Cache::close` function to release system resources when you are finished using the cache. After the cache is closed, any further method calls on the cache or any region object results in a `CacheClosedException`.

If the cache is in a durable client, you need to use the `keepalive` version of the `close` method. See *Disconnecting From the Server* on page 156.

2.3 Regions

You can create cache regions either programmatically or through declarative statements in the `cache.xml` file. Generally, a cache is organized and populated through a combination of the two approaches.

Region creation is subject to attribute consistency checks. A distributed region can be either non-partitioned or a partitioned region. See the *GemFire Enterprise Developer's Guide* for detailed descriptions of both non-partitioned and partitioned regions. The requirements for consistency between attributes are detailed both in the online API documentation and throughout the discussion of *Region Attributes* on page 53.

The following topics describe several scenarios for creating and accessing regions.

Declarative Region Creation

Declarative region creation involves placing the region's XML declaration, with the appropriate attribute settings, in the `cache.xml` file that is loaded at cache creation. Regions are placed inside the `cache` declaration in `region` elements.

Example 2.1 Declarative Region Creation for a Native Client

```
<cache>
   <pool name="examplePool" subscription-enabled="true" >
       <server host="localhost" port="40404" />
   </pool>
   <region name="A" refid="PROXY">
     <region-attributes pool-name="examplePool"/>
   </region>
   <region name="A1">
     <region-attributes refid="PROXY" pool-name="examplePool"/>
   </region>
   <region name="A2" refid="CACHING_PROXY">
      <region-attributes pool-name="examplePool"/>
        <region-time-to-live>
          <expiration-attributes timeout="120" action="invalidate"/>
        </region-time-to-live>
     </region-attributes>
   </region>
</cache>
```

The `cache.xml` file contents must conform to the DTD provided in the *`productDir`*`/dtd/gfcpp-cache3500.dtd` file. For details, see *Cache Initialization File* on page 75.

Programmatic Region Creation

Create your regions using the `regionFactory` class:

Example 2.2 C++ RegionFactory

```
RegionFactoryPtr regionFactory =
   cachePtr->createRegionFactory(CACHING_PROXY);
RegionPtr regPtr0 = regionFactory->setLruEntriesLimit(20000)
                 ->create("exampleRegion0");
```

For details, see the online API documentation.

Region Access

You can use `Cache::getRegion` to retrieve a reference to a specified region. `RegionPtr` returns `NULL` if the region is not already present in the application's cache. A server region must already exist.

A region's name cannot contain the slash character. Region names also cannot contain these characters:

```
<
>
:
"
\
|
?
*
```

Getting the Region Size

The `Region` API provides a `size` method (`Size` property for .NET) that gets the size of a region. For native client regions, this gives the number of entries in the local cache, not on the servers.

See the `Region` online API documentation for details.

Invalidating or Destroying a Region

Invalidation marks all entries contained in the region as invalid (with null values). Destruction removes the region and all of its contents from the cache. These operations may be carried out explicitly in the local cache in the following ways:

- Through direct API calls from the native client.
- Through expiration activities based on the region’s statistics and attribute settings.

In either case, invalidation and destruction may be performed as a local or a distributed operation:

- A local operation affects the region only in the local cache.
- A distributed operation works first on the region in the local cache and then distributes the operation to all other caches where the region is defined. This is the proper choice when the region is no longer needed, or valid, for any application in the distributed system.

Note that a user-defined cache writer can abort a region destroy operation. Cache writers are synchronous listeners with the ability to abort operations. If a cache writer is defined for the region anywhere in the distributed system, it is invoked before the region is explicitly destroyed.

Region invalidation and destruction can cause other user-defined application plug-ins to be invoked as well. These plug-ins are described in detail in *Application Plug-Ins* on page 61.

Whether carried out explicitly or through expiration activities, invalidation and destruction cause event notification.

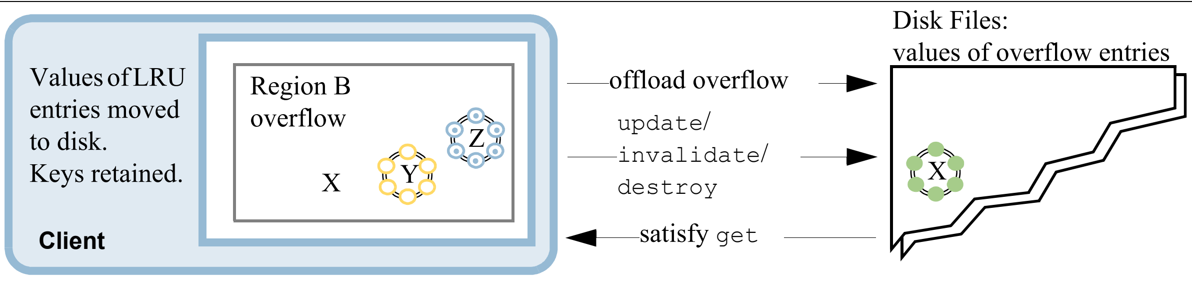
Disk Storage for Region Data

Region data can be stored to disk using an overflow process to satisfy region capacity restrictions without completely destroying the local cache data. The storage mechanism uses disk files to hold region entry data. When an entry is overflowed, its value is written to disk but its key and entry object remain in the cache.

Overflow allows you to keep the region within a user-specified size in memory by relegating the values of least recently used (LRU) entries to disk. Overflow essentially uses disk as a swap space for entry values. When the region size reaches the specified threshold, entry values are moved from memory to disk, as shown in the following figure. If an entry is requested whose value is only on disk, the value is copied back into memory, possibly causing the value of a different LRU entry to be overflowed to disk.

See *Cache Eviction and Overflow* on page 88 for more information about data overflow methods. Also, see *DiskPolicy* on page 57 for details about the related `disk-policy` attribute.

Figure 2.1 Data Flow Between Overflow Region and Disk Files


Values of LRU entries moved to disk.
Keys retained.
Client
Region B overflow
X
Y
Z
offload overflow
update/ invalidate/ destroy
satisfy get
Disk Files: values of overflow entries
X

In this figure the value of the LRU entry X has been moved to disk to recover space in memory. The key for the entry remains in memory. From the distributed system perspective, the value on disk is as much a part of the region as the data in memory. A `get` *performed on region B looks first in memory and then on disk as part of the local cache search.*

2.4 Entries

Region entries hold the cached application data. Entries are automatically managed according to region attribute settings, and can be created, updated, invalidated, and destroyed through explicit API calls or through operations distributed from other caches. When the number of entries is very large, a partitioned region can provide the required data management capacity if the total size of the data is greater than the heap in any single VM. See the *GemFire Enterprise Developer's Guide* for a detailed description of partitioned regions and related data management.

When an entry is created, a new object is instantiated in the region containing:

- The entry key.
- The entry value. This is the application data object. The entry value may be set to `NULL`, which is the equivalent of an invalid value.

Entry operations invoke callbacks to user-defined application plug-ins. In this section, the calls are highlighted that may affect the entry operation itself (by providing a value or aborting the operation, for example), but all possible interactions are not listed. For details, see *Application Plug-Ins* on page 61.

`DateTime` objects must be stored in the cache in UTC, so that times correspond between client and server. If you use a date with a different time zone, convert it when storing into and retrieving from the cache.

Requirements for Distribution

For entry data distributed between members of the distributed system, the following requirements apply:

- All data must be serializable. Entry keys and values are serialized for distribution.
- If a native client defines a region, it must register any serializable types for all the classes of objects stored in the region. This includes entries that the application gets or puts, as well as entries that are pushed to the client's cache automatically through distribution. The types must be registered before the native client connects to the distributed system.

See *Serialization* on page 93 for more information about these requirements.

Registering Interest for Entries

Native client regions can programmatically register interest in entry keys stored on a cache server region. Interest can be registered for specific entry keys or for all keys, and regular expressions can be used to register interest for keys whose strings match the expression.

By registering interest, a client region receives update notifications from the cache server for the keys of interest. Clients can also unregister interest for specific keys, groups of keys based on regular expressions, or for all keys.

Interest registration and unregistration are symmetrical operations. Consequently, you cannot register interest in all keys and then unregister interest a specific set of keys. Also, if you first register interest in specific keys with `registerKeys`*, then call* `registerAllKeys`*, you must call* `unregisterAllKeys` *before specifying interest in specific keys again.*

Client API for Registering Interest

Client interest registration can be accomplished through both the C++ and the .NET API. The C++ API provides the `registerKeys`, `registerAllKeys`, and `registerRegex` methods, with corresponding unregistration accomplished using the `unregisterKeys`, `unregisterAllKeys`, and `unregisterRegex` methods. The .NET API provides the `RegisterKeys`, `RegisterAllKeys`, and

`RegisterRegex` methods, with corresponding unregistration accomplished using the `UnregisterKeys`, `UnregisterAllKeys`, and `UnregisterRegex` methods.

The `registerkeys`, `registerRegex` and `registerAllKeys` methods have the option to populate the cache with the registration results from the server. The `registerRegex` and `registerAllKeys` methods can also optionally return the current list of keys registered on the server

Client Notification

In addition to the programmatic function calls, to register interest for a server region and receive updated entries you need to configure the region with `PROXY` or `CACHING_PROXY` `RegionShortcut` setting. Otherwise, when you register interest, you'll get an `UnsupportedOperationException`.

Example 2.3 Enabling client-notification in the Native Client cache.xml File

```
<region name = "listenerWriterLoader" refid="CACHING_PROXY">
   ...
```

Both native clients and Java clients that have subscriptions enabled track and drop (ignore) any duplicate notifications received. To reduce resource usage, a client expires tracked sources for which new notifications have not been received for a configurable amount of time.

Notification Sequence

Notifications invoke CacheListeners of cacheless clients in all cases for keys that have been registered on the server. Similarly, invalidates received from the server invoke CacheListeners of cacheless clients.

If you register to receive notifications, listener callbacks are invoked irrespective of whether the key is in the client cache when a `destroy` or `invalidate` event is received.

Registering Interest for Specific Keys

Registering and unregistering interest for specific keys is accomplished using the `registerKeys` and `unregisterKeys` functions. You register interest in a key or set of keys by specifying the key name using the programmatic syntax shown in the following example:

Example 2.4 Programmatically Registering Interest in a Specific Key

```
keys0.push_back(keyPtr1);
keys1.push_back(keyPtr3);
regPtr0->registerKeys(keys0);
regPtr1->registerKeys(keys1);
```

The programmatic code snippet in the next example shows how to unregister interest in specific keys:

Example 2.5 Programmatically Unregistering Interest in a Specific Key

```
regPtr0->unregisterKeys(keys0);
regPtr1->unregisterKeys(keys1);
```

Registering Interest for All Keys

If the client registers interest in all keys, the server provides notifications for all updates to all keys in the region. The next example shows how to register interest in all keys:

Example 2.6 Programmatically Registering Interest in All Keys

```
regPtr0->registerAllKeys();
regPtr1->registerAllKeys();
```

The following example shows a code sample for unregistering interest in all keys.

Example 2.7 Programmatically Unregistering Interest in All Keys

```
regPtr0->unregisterAllKeys();
regPtr1->unregisterAllKeys();
```

Registering Interest Using Regular Expressions

The `registerRegex` function registers interest in a regular expression pattern. The server automatically sends the client changes for entries whose keys match the specified pattern.

Keys must be strings in order to register interest using regular expressions.

The following example shows interest registration for all keys whose first four characters are `Key-`, followed by any string of characters. The characters `.*` represent a wildcard that matches any string.

Example 2.8 Programmatically Registering Interest Using Regular Expressions

```
regPtr1->registerRegex("Key-.*");
```

To unregister interest using regular expressions, you use the `unregisterRegex` function. The next example shows how to unregister interest in all keys whose first four characters are `Key-`, followed by any string (represented by the `.*` wildcard).

Example 2.9 Programmatically Unregistering Interest Using Regular Expressions

```
regPtr1->unregisterRegex("Key-.*");
```

Register Interest Scenario

In this register interest scenario, a cache listener is used with a cacheless region that has `client-notification` set to `true`. The client region is configured with caching disabled, client notification is enabled, and a cache listener is established. The client has not registered interest in any keys:

When a value changes in another client, it sends the event to the server. The server will not send the event to the cacheless client, even though `client-notification` is set to `true`, unless the client has previously performed a distributed operation on that key.

To activate the cache listener so the cacheless region receives updates, the region should register interest in some or all keys by using one of the API calls for registering interest. This way, the client receives all events for the keys to which it has registered interest.

Using serverKeys to Retrieve a Set of Region Keys

You can retrieve the set of keys defined in the cache server process that are associated with the client region by using the `Region::serverKeys` API function. If the server region is defined as a replicate, then the keys returned will consist of the entire set of keys for the region.

The following example shows how the client can programmatically call `serverKeys`.

Example 2.10 Using serverKeys to Retrieve the Set of Keys From the Server

```
VectorOfCacheableKey keysVec;
region->serverKeys( keysVec );
size_t vlen = keysVec.size();
bool foundKey1 = false;
bool foundKey2 = false;
for( size_t i = 0; i < vlen; i++ ) {
    CacheableStringPtr strPtr = dynCast<CacheableStringPtr>( keysVec.at( i );
    std::string veckey = strPtr->asChar();
    if ( veckey == "skey1" ) {
        printf( "found skey1" );
        foundKey1 = true;
    }
    if ( veckey == "skey2" ) {
        printf( "found skey2" );
        foundKey2 = true;
    }
}
```

An `UnsupportedOperationException` occurs if the client region is not a native client region. A `MessageException` occurs if the message received from the server could not be handled, which can occur if an unregistered `typeId` is received in the reply.

Adding Entries to the Cache

A region is populated with cached entries in several ways:

- Explicitly, when an application executes a `create` or a `put` operation for a single entry or for multiple entries that do not already exist in the cache.
- Implicitly, when a client does a `get` on a single entry or on multiple entries that do not already exist in the cache. In this case, the entry is retrieved from a remote cache or through a cache loader (see *CacheLoader* on page 62). A client can also use `getAll` to populate a region with all values for an array of keys. See *Accessing Entries* on page 50.
- Automatically, when entries are created in remote caches.

If any cache writer is available in the distributed region, it is called before the entry is created and it can abort the creation process.

Adding Entries to the Local Cache

If just the local cache is to be populated, you can either `create` an entry using the `localCreate` Region API, or `put` an entry using `localPut`. See the C++ and .NET online API documentation for details about `localCreate` and `localPut`.

`DateTime` objects must be stored in the cache in UTC, so that times correspond between client and server. If you use a date with a different time zone, convert it when storing into and retrieving from the cache. This example converts a local time to UTC for a put operation:

```
DateTime t1( 2009, 8, 13, 4, 11, 0, DateTimeKind.Local);
region0.Put( 1, t1.ToUniversalTime() );
```

Adding Multiple Entries Using PutAll

If you have a large number of cache entries to add to a region at one time, you can improve cache performance by using the `putAll` function to add them in a single distributed operation. Multiple key/value pairs are stored in a hashmap, then the hashmap contents are processed on the server as either new entries, updates, or invalidates for existing entries.

See *Bulk Put Operations Using putAll* on page 91 for more information about the `putAll` API. Additional details are available in the online API documentation for `Region::putAll` (C++), or `Region.PutAll` (.NET).

Updating Entries

A cached entry can be updated using these methods:

- Explicitly, when a client invokes a `put` operation on an existing entry.
- Implicitly, when a `get` is performed on an entry that has an invalid value in the cache. An entry can become invalid through an explicit API call, through an automated expiration action, or by being created with a value of `null`.
- Automatically, when a new entry value is distributed from another cache.

Similar to entry creation, all of these operations can be aborted by a cache writer.

The `get` function returns a direct reference to the entry value object. A change made using that reference is called an in-place change because it directly modifies the contents of the value in the local cache. For details on safe cache access, see *Changed Objects* on page 71.

Accessing Entries

The API provides operations for the retrieval of the entry key, entry value, and the `RegionEntry` object itself. A variety of functions provide information for individual entries and for the set of all entries resident in the region. The online API documentation lists all available access functions.

A region's entry keys and `RegionEntry` objects are directly available from the local cache. Applications can directly access the local cache's stored entry value through the `RegionEntry::getValue` function. The `getValue` function either returns the value if a valid value is present in the local cache, or `NULL` if the value is not valid locally. This function does no data loading, nor does it look elsewhere in the distributed system for a valid value.

> *Direct access through* `RegionEntry::getValue` *does not reset an entry's timestamp for LRU expiration. See Expiration Attributes* on page 69 *for more information about LRU expiration.*

In comparison, the standard `Region::get` functions consider all caches and all applicable loaders in the distributed system in an attempt to return a valid entry value to the calling application. The primary attribute setting affecting entry retrieval is CacheLoader (page 62).

The standard `Region::get` functions may implement a number of operations in order to retrieve a valid entry value. The operations used depend on the region's attribute settings and on the state of the entry itself. By default, the client retrieves entry values through calls to the `get` function. The client can override this behavior for any region by defining a cache loader for the region.

The following sections discuss the `get` function and special considerations for entry retrieval.

Entry Retrieval

Entry values are retrieved through the `Region::get` function.

When an entry value is requested from a region, it is either retrieved from the cache server or fetched by the region's locally-defined cache loader in this sequence:

1. local cache search
2. server cache
3. local load (For distributed regions, the local load is favored over remote cache values)

How the get Operation Affects the Local Entry Value

If a `get` operation retrieves an entry value from outside the local cache through a local load, it automatically `puts` the value into the cache for future reference.

Note that these load operations do not invoke a cache writer. Because the loader and writer operate against the same data source, there is no need to perform a cache write for entries that were just fetched from that data source. For descriptions of these processes, see *Application Plug-Ins* on page 61.

Access through a `get` *operation resets an entry's timestamp for LRU expiration.*

Getting Multiple Entries Using getAll

You can use the `getAll` Region API to get all values for an array of keys from the local cache or cache server. See *Bulk Get Operations Using getAll* on page 92 for more information.

Invalidating or Destroying Cached Entries

Invalidating an entry sets the entry's value to `NULL`. Destroying it removes the entry from the region altogether. These operations can be carried out in the local cache in the following ways:

- Through direct API calls from the client.
- Through expiration activities based on the entry's statistics and the region's attribute settings.

A user-defined cache writer is called before an operation is completed, and can abort an entry destroy operation.

Whether carried out explicitly or through expiration activities, invalidation and destruction cause event notification: The `CacheEvent` object has an `isExpiration` flag that is set to `true` for events resulting from expiration activities, and set to `false` for all other events.

Notification for Operations

Listeners are invoked for client-initiated operations only after the client operation succeeds on the server. Listener invocation on the client indicates that the server has the same data as the client.

If a client operation fails on the server then the operation is rolled back, assuming that no other thread has modified the data in the intervening period. Rollback may not be possible in cases where the entry has been evicted by LRU or expiration during this period, so when an exception is received from the server for an operation then local changes may not have been rolled back

Event Notification Sequence

Events received on the clients that originated on the server invoke the subscription for the event as seen by the server. Events originating on the client invoke the subscription as seen by the client.

For example, a client that receives a `create` and an `update` from the server fires a `create` event and an `update` event because that is how the server saw it. A cacheless client that does two consecutive `put` operations has two `afterCreate` events invoked on the originating client because the client does not have any history about the first `put`, since it is cacheless.

For the same sequence, the server sees an `afterCreate` and an `afterUpdate` event, and a remote client receiving the event sees an `afterCreate` followed by an `afterUpdate` event. A client that caches locally sees an `afterCreate` and an `afterUpdate` for the same scenario (as will the server and remote clients).

2.5 Region Attributes

Region attributes govern the automated management of a region and its entries. Region attributes determine such things as where the data resides, how the region is managed in memory, and the automatic loading, distribution, and expiration of region entries.

Specify region attributes before creating the region. You can do this either through the declarative XML file or through the API. The API includes classes for defining a region's attributes before creation and for modifying some of them after creation. For details, see the online API documentation for `RegionShortcut`, `RegionAttributes`, `AttributesFactory`, and `AttributesMutator`.

RegionShortcuts

GemFire provides a number of predefined, shortcut region attributes settings for your use, in `RegionShortcut`. You can also create custom region attributes and store them with an identifier for later retrieval. Both types of stored attributes are referred to as named region attributes. You can create and store your attribute settings in the `cache.xml` file and through the API.

Retrieve named attributes by providing the ID to the region creation. This example uses the shortcut `CACHING_PROXY` attributes to create a region:

Example 2.11 Using a Region Shortcut to Define a Region

```
<region name="testRegion" refid="CACHING_PROXY"/>
```

You can modify named attributes as needed. For example, this adds a cache listener to the region:

Example 2.12 Using a Region Shortcut with Overrides

```
<region name="testRegion" refid="CACHING_PROXY">
   <region-attributes>
      <cache-listener library-name="myAppLib"
         library-function-name ="myCacheListener" />
   </region-attributes>
</region>
```

In this example, the modified region shortcut is saved to the cache using the region attribute `id`, for retrieval and use by a second region:

Example 2.13 Storing and Using a New Named Region Attributes Setting

```
<region name="testRegion" refid="CACHING_PROXY">
   <region-attributes id="Caching_Proxy_With_Listener">
      <cache-listener library-name="myAppLib"
         library-function-name ="myCacheListener" />
   </region-attributes>
</region>
<region name="newTestRegion" refid="Caching_Proxy_With_Listener"/>
```

Shortcut Attribute Options

You can select the most common region attributes settings from `RegionShortcut`, GemFire's predefined named region attributes.

Shortcut attributes are a convenience only. They are simply named attributes that GemFire has already stored for you. You can override their settings by storing new attributes with the same `id` as the predefined attributes.

This section provides an overview of the options available in the region shortcut settings.

For complete listings and descriptions, including information on the underlying `RegionAttributes` *settings for each shortcut, see the online documentation for* `RegionShortcut`*.*

These are the options available in `RegionShortcut`.

Communication with Servers and Data Storage

- `PROXY` does not store data in the client cache, but connects the region to the servers for data requests and updates, interest registrations, and so on.
- `CACHING_PROXY` stores data in the client cache and connects the region to the servers for data requests and updates, interest registrations, and so on.
- `LOCAL` stores data in the client cache and does not connect the region to the servers. This is a client-side-only region.

Data Eviction

For the non-`PROXY` regions—the regions that store data in the client cache—you can add data eviction:

- `ENTRY_LRU` causes least recently used data to be evicted from memory when the region reaches the entry count limit.

Region Attributes Descriptions

Attributes that are immutable (fixed) after region creation govern such things as storage location, data distribution, statistics, application plug-ins, and the configuration and management of the region's data hashmap.

This table lists the immutable attributes and their default settings.

Table 2.1 Immutable Region Attributes

Immutable Region Attribute	Default Setting
CachingEnabled (page 56)	`true`
InitialCapacity (page 56)	`16` (entries)
LoadFactor (page 56)	`0.75`
ConcurrencyLevel (page 56)	`16`
LruEntriesLimit (page 57)	
DiskPolicy (page 57)	
PersistenceManager (page 57)	`NULL`
PartitionResolver (page 64)	

The remaining region attributes identify expiration and cache listener, cache writer and cache loader actions that are run from the defining client. The next table lists the mutable attributes that generally may be modified after region creation by using the `AttributesMutator` for the region.

Table 2.2 Mutable Region Attributes

Mutable Region Attribute	Default Setting
Expiration Attributes (page 69)	no expiration
LruEntriesLimit (page 57)	`0` (no limit)
CacheLoader (page 62)	
CacheWriter (page 62)	
CacheListener (page 62)	

See *Using AttributesMutator to Modify a Plug-In* on page 67 for information about using `AttributesMutator` with cache listeners, cache loaders and cache writers.

The remainder of this section examines these attributes in detail. Throughout the descriptions, `cache.xml` file snippets show how each attribute can be set declaratively.

CachingEnabled

This attribute determines whether data is cached in this region. For example, you might choose to configure the distributed system as a simple messaging service where clients run without a cache.

> *You can configure the most common of these options with GemFire's predefined region attributes. See RegionShortcuts* on page 53 *and the Javadocs for* `RegionShortcut`.

If this attribute is `false` (no caching), an `IllegalStateException` is thrown if any of these attributes are set:

- `InitialCapacity` (page 56)
- EntryTimeToLive (page 69)
- EntryIdleTimeout (page 69)
- `LoadFactor` (page 56)
- `ConcurrencyLevel` (page 56)
- `LruEntriesLimit` (page 57)
- `DiskPolicy` (page 57)

The following declaration enables caching for the region:

```
<region-attributes caching-enabled="true">
</region-attributes>
```

InitialCapacity

This attribute, together with the `LoadFactor` attribute, sets the initial parameters on the underlying hashmap used for storing region entries. This is the entry capacity that the region map will be ready to hold when it is created.

This declaration sets the region's initial capacity to `10000`:

```
<region-attributes initial-capacity="10000">
</region-attributes>
```

LoadFactor

This attribute, together with the `InitialCapacity` attribute, sets the initial parameters on the underlying hashmap used for storing region entries. When the number of entries in the map exceeds the `LoadFactor` times current capacity, the capacity is increased and the map is rehashed. The best performance is achieved if you configure a properly sized region at the start and do not have to rehash it.

This declaration sets the region's load factor to `0.75`:

```
<region-attributes load-factor="0.75">
</region-attributes>
```

ConcurrencyLevel

This attribute estimates the maximum number of application threads that concurrently access a region entry at one time. This attribute helps optimize the use of system resources and reduce thread contention.

The following declaration sets the region's `ConcurrencyLevel` to `16`:

```
<region-attributes concurrency-level="16">
</region-attributes>
```

When `CachingEnabled` *is* `false`*, do not set the* `ConcurrencyLevel` *attribute. An* `IllegalStateException` *is thrown if the attribute is set.*

LruEntriesLimit

This attribute sets the maximum number of entries to hold in a caching region. When the capacity of the caching region is exceeded, a least-recently-used (LRU) algorithm is used to evict entries.

This is a tuning parameter that affects system performance.

This declaration limits the region to 20,000 entries:

```
<region-attributes lru-entries-limit="20000"
      initial-capacity="20000"
      load-factor="1">
</region-attributes>
```

Evicted entries can be destroyed or moved to disk as an extension of the cache. See DiskPolicy (page 57).

When `CachingEnabled` *is* `false`*, do not set the* `LruEntriesLimit` *attribute. An* `IllegalStateException` *is thrown if the attribute is set.*

DiskPolicy

If the `lru-entries-limit` attribute is greater than zero, the optional `disk-policy` attribute determines how over-limit LRU entries are handled. LRU entries over the limit are either destroyed by default (`disk-policy` is `none`) or written to disk (`overflows`).

The following declaration sets the region's LRU eviction policy to overflow to disk:

```
<region-attributes lru-entries-limit="nnnnn"
         disk-policy="overflows">
   <persistence-manager .../>
</region-attributes>
```

If `disk-policy` is set to `overflows`, a persistence manager must be specified to implement the actual disk writes and reads. The persistence manager provides the interface to the back-end disk or database. See `PersistenceManager` (page 57) for more information.

If `LruEntriesLimit` *is* `0`*, or* `CachingEnabled` *is* `false`*, do not set the* `disk-policy` *attribute. An* `IllegalStateException` *is thrown if the attribute is set.*

PersistenceManager

For each region, if the `disk-policy` attribute is set to `overflows` then a `persistence-manager` plug-in performs the cache-to-disk and disk-to-cache operations. See *Application Plug-Ins* on page 61.

The persistence manager plug-in attributes are declared as follows:

```
<region-attributes lru-entries-limit="nnnnn"
      disk-policy="overflows">
   <persistence-manager library-name="libraryName"
         library-function-name="functionName">
      <properties>
         <property name="propertyName" value="propertyValue" />
      </properties>
   </persistence-manager>
</region-attributes>
```

The optional properties set parameters for the plug-in.

Berkeley DB Persistence Manager

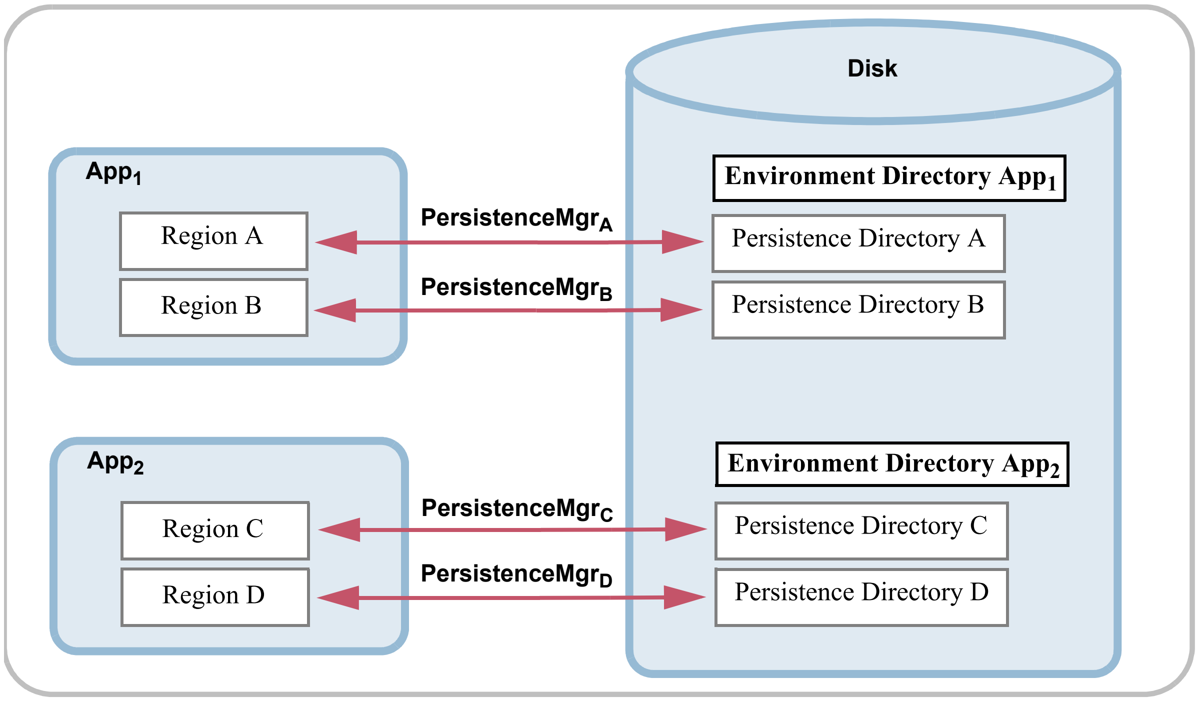
The persistence manager that is provided for the GemFire native client uses the open-source Berkeley DB (BDB) library. This section describes BDB configuration.

> *For information about downloading, building and installing the BDB library, see* Appendix A, *Installing the Berkeley DB Persistence Manager*, on page 289.

Each BDB persistence manager (PM) stores its region data in a Berkeley database that is stored in disk files, as shown in Figure 2.2 on page 58.

- Each process must have a unique environment (DB) directory. See `EnvironmentDirectory` (page 59).
- In a process, each region's BDB persistence manager shares a single Berkeley database environment directory. Each region must have a unique persistence (overflow) directory. See `PersistenceDirectory` (page 59).

Figure 2.2 Berkeley Database Persistence Manager Directory Structure



The BDB persistence manager has the following attributes:

```
<region-attributes>
 <persistence-manager library-name="libBDBImpl.so"
                  library-function-name="createBDBInstance">
  <properties>
   <property name="PersistenceDirectory" value="/xyz"/>
   <property name="EnvironmentDirectory" value="/home/abc"/>
   <property name="CacheSizeGb" value="1"/>
   <property name="CacheSizeMb" value="0"/>
   <property name="PageSize" value="65536"/>
   <property name="MaxFileSize" value="512000000"/>
  </properties>
 </persistence-manager>
</region-attributes>
```

Table 2.3 on page 59 describes the BDB directory location attributes.

Table 2.4 on page 60 describes the BDB size tuning attributes.

To set these attributes programmatically, see *Handling Cache LRU* on page 89.

Table 2.3 Berkeley DB Persistence Manager Directory Attributes

Property	Description	Default Setting
`Environment Directory`	Directory where Berkeley DB environment files are created. This setting must be different for each process. The environment is initialized by the first region that instantiates and initializes a persistence manager (PM). Each subsequent region in the same process must use the same setting for its persistence manager or that PM will fail. This directory is created by the first persistence manager. The persistence manager initialization will fail if the directory already exists or could not be created.	`BDBImplEnv` in the directory where the process was started.
`Persistence Directory`	Directory where each region's DB files are stored. This setting must be different for each region, including regions in different processes. This directory is created by the persistence manager. The persistence manager initialization will fail if the directory already exists or could not be created.	`BDBImplRegionData_`*`regionName`* in the directory where the process was started.

Berkeley DB performance is optimized by configuring the environment and database with the settings listed in Table 2.4 on page 60. The recommended settings are provided as default values.

Berkeley databases share a memory pool of database pages. The environment cache is a major factor that affects DB performance. It is recommended that the cache size be set as large as possible without causing performance degradation in the virtual memory subsystem.

Table 2.4 Berkeley DB Persistence Manager Database Tuning Attributes

Property	Description	Default Setting
`CacheSizeGb`	Size of DB environment cache, in gigabytes. This is added to `CacheSizeMb`.	`0`
`CacheSizeMb`	Size of DB environment cache, in megabytes. This is added to `CacheSizeGb`.	`300`
`PageSize`	Database page size, in bytes. Disk operations are performed at the granularity of a database page. Performance is also affected as locking is performed at the page level as the persistence manager initializes all databases with the `BTree` access method. `PageSize` should be reduced if there is significant performance degradation in multithreaded applications.	`65536`
`MaxFileSize`	Maximum database file size, in bytes. The persistence manager stores data for a particular region in a number of databases, each of which is stored in a separate file. As data is added to the database, the performance of database operations degrades after a certain database size. By ensuring that each database file does not grow beyond a certain limit, performance of read/write operations can be preserved as the number of entries in the region grows. The default value is recommended for entry sizes of 1Kb. As average entry size reduces, the limit can be increased.	`512000000`

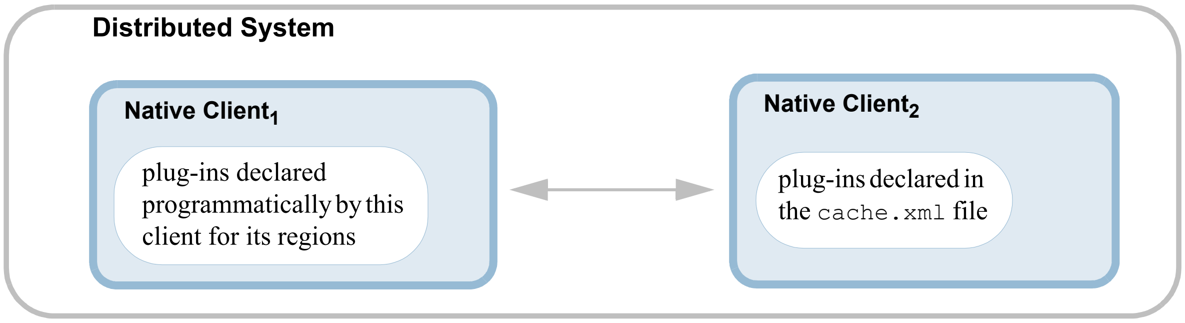
See the Berkeley DB documentation at http://www.oracle.com/database/berkeley-db/index.html for more details on database tuning.

Application Plug-Ins

The plug-in attributes allow you to customize client region behavior for loading, updating, deleting, and overflowing region data and for accessing data in server partitioned regions. All client plug-ins are available through the C++ and .NET API.

Application plug-ins for cache regions in clients can be declared either programmatically or in the `cache.xml` file.

Figure 2.3 Where Application Plug-Ins Run


Distributed System
Native Client1
plug-ins declared programmatically by this client for its regions
Native Client2
plug-ins declared in the cache.xml file

The following XML declaration specifies a cache loader for a region when the region is created.

Example 2.14 Declaring a Cache Loader When a Region is Created

```
<region-attributes>
   <cache-loader library-name="appl-lib"
         library-function-name ="createCacheLoader">
   </cache-loader>
</region-attributes>
```

The API provides the framework for application plug-ins with callback functions for the appropriate events. Your classes and functions can customize these for your application's needs. When creating a region, specify these as part of the region's attributes settings. For regions already in the cache, you can specify new `CacheLoader`, `CacheWriter`, and `CacheListener` using the region's `AttributesMutator`. The `PartitionResolver` is not mutable.

- **`CacheLoader`**—A data loader called when an entry `get` operation fails to find a value for a given key. A cache loader is generally used to retrieve data from an outside source such as a database, but it may perform any operation defined by the user. Loaders are invoked as part of the distributed loading activities for entry retrieval, described in *Entry Retrieval* on page 51.
- **`CacheWriter`**—A synchronous event listener that receives callbacks before region events occur and has the ability to abort the operations. Writers are generally used to keep a back-end data source synchronized with the cache.
- **`CacheListener`**—An asynchronous event listener for region events in the local cache.
- **`PartitionResolver`**—Used for single-hop access to partitioned region entries on the server side. This resolver implementation must match that of the `PartitionResolver` on the server side.

The rest of this section gives more detailed descriptions of these application plug-ins, followed by special considerations for plug-ins in distributed regions and some guidelines for writing callbacks.

CacheLoader

A cache loader is an application plug-in used to load data into the region. When an entry is requested that is unavailable in the region, a cache loader may be called upon to load it. Generally, a cache loader is used to retrieve the data from a database or some other source outside the distributed system, but it may perform any operation defined by the user.

The `CacheLoader` interface provides one function, `load`, for customizing region entry loading. A distributed region may have cache loaders defined in any or all caches where the region is defined. When loading an entry value, a locally defined cache loader is always used before a remote loader. In distributed regions, loaders are available for remote entry retrieval.

CacheWriter

A cache writer is an application plug-in used to synchronously handle changes to a region's contents. It is generally used to keep back-end data sources synchronized with a cache region. A cache writer has callback functions to handle region destruction and entry creation, update, and destruction. These functions are all called before the modification has taken place and can abort the operation.

You can also use cache writers to store data that you want to make persistent.

CacheListener

A cache listener is an application plug-in used to asynchronously handle changes to a region's contents. A cache listener has callback functions to handle region destruction and invalidation, along with entry creation, update, invalidation, and destruction. These functions are called asynchronously after the modification has taken place.

This declarative XML example establishes a cache listener when a region is created:

Example 2.15 Declaring a Cache Listener When a Region is Created

```
<region name=”region11”>
   <region-attributes>
      <cache-listener library-name="appl-lib"
         library-function-name ="createCacheListener" />
   </region-attributes>
</region>
```

Unlike cache loaders and cache writers, cache listeners only receive events for entries to which the client has performed operations or registered interest.

When the listener is attached to a region with caching disabled, the old value is always `NULL`.

Caution: Do not perform region operations inside the cache listener. Once you have configured a cache listener, the event supplies the new entry values to the application. Performing a `get` *with a key from the* `EntryEvent` *can result in distributed deadlock. For more about this, see the online API documentation for* `EntryEvent`.

When a region disconnects from a cache listener, you can implement the `afterRegionDisconnected` callback event. This callback event is only be invoked when using the `pool` API and `subscription` is enabled on the pool. For example:

Example 2.16 Defining Callback After Region Disconnects From Cache Listener

```
class DisconnectCacheListener : public CacheListener
{
   void afterRegionDisconnected( const RegionPtr& region )
   {
      printf("After Region Disconnected event received");
   }
};
```

PartitionResolver

This section pertains to data access in server regions that have custom partitioning. Custom partitioning uses a Java `PartitionResolver` to colocate like data in the same buckets. For the client, you can use a `PartitionResolver` that matches the server's implementation to access data in a single hop. With single-hop data access, the client pool maintains information on where a partitioned region's data is hosted. When accessing a single entry, the client directly contacts the server that hosts the key—in a single hop.

> *Single hop is only used for these operations that are run on single data points:* `put`, `get`, `getAll`, `destroy`.

To use single hop on a partitioned region:

1. Make sure the pool attribute, `pr-single-hop-enabled` (page 238) is set to `true` or not set. It is `true` by default.
2. If the server uses a custom `PartitionResolver` install an implementation of `PartitionResolver` in the client region that returns, entry for entry, the same value as the server's Java `PartitionResolver` implementation. The server uses the resolver to colocate like data within a partitioned region.

   If the server does not use a custom resolver, the default resolvers in client and server match, so single hop will work there by default.

Disable single hop behavior for a region by setting its pool's attribute, `pr-single-hop-enabled` (page 209) to `false`.

To disable single hop behavior, set the pool attribute, `pr-single-hop-enabled` (page 238), to `false`.

See the *GemFire Enterprise Developer's Guide* for more information, including colocating like data within a partitioned region and how to get get the best performance with PR single hop.

Implementing a PartitionResolver

See the *GemFire Enterprise Developer's Guide* for information on custom partitioning the server partitioned regions.

Your implementation of `PartitionResolver` must match that of the server side.

1. Implement `PartitionResolver` in the same place as you did in the server—custom class, key, or cache callback argument.
2. Program the resolver's functions the same way you programmed them in the Java implementation. Your implementation must match the server's.

Example 2.17 C++ - Programming the PartitionResolver

```
class TradeKeyResolver : public PartitionResolver
{
private:
   string m_tradeID;
   int m_month;
   int m_year;
public:
   TradeKeyResolver() { }
   TradeKeyResolver(int month, int year) {
      m_month = month;
      m_year = year;
   }
   ~TradeKeyResolver() { }
```

```
            static PartitionResolverPtr createTradeKeyResolver() {
               PartitionResolverPtr tradeKeyResolver( new TradeKeyResolver());
            return tradeKeyResolver;
            }
            const char* getName() {
               return "TradeKey";
            }
            CacheableKeyPtr getRoutingObject(const EntryEvent& opDetails) {
                return CacheableKey::create(m_month + m_year);
            }
         };
```

Example 2.18 Csharp - Programming the PartitionResolver

```
using System;
using System.Threading;
// Use the GemFire namespace
using GemStone.GemFire.Cache;
class TradeKeyResolver : IPartitionResolver
{
   private int m_month = 0;
   private int m_year = 0;

   public static TradeKeyResolver CreateTradeKeyResolver()
   {
      return new TradeKeyResolver();
   }

   public virtual ICacheableKey GetRoutingObject(EntryEvent entry)
   {
      return new CacheableInt32(m_month + m_year);
   }

   public virtual String GetName()
   {
      return "TradeKeyResolver";
   }
}
```

3. Install the resolver in the region. Use one of these methods:

Example 2.19 XML partition resolver declaration

```
<region name="trades" refid="CACHING_PROXY">
   <region-attributes>
      <partition-resolver library-name="appl-lib" library-function-name=
      "createTradeKeyResolver"/>
   </region-attributes>
</region>
<pool free-connection-timeout="12345" idle-timeout="5555"
      load-conditioning-interval="23456" max-connections="7"
      min-connections="3" name="test_pool_1" ping-interval="12345"
      read-timeout="23456" retry-attempts="3" server-group="ServerGroup1"
      socket-buffer-size="32768" statistic-interval="10123"
```

```
            subscription-ack-interval="567" subscription-enabled="true"
            subscription-message-tracking-timeout="900123"
            subscription-redundancy="0" thread-local-connections="5"
            pr-single-hop-enabled="true">
        <locator host="localhost" port="34756"/>
    </pool>
```

Example 2.20 Programmatic partition resolver installation

```
    void setPartitionResolver()
    {
        CachePtr cachePtr = CacheFactory::createCacheFactory()->create();
        PartitionResolverPtr resolver( new TradeKeyResolver());
        RegionFactoryPtr regionFactory =
            cachePtr->createRegionFactory(PROXY)
                ->setClientNotificationEnabled(true)
                ->setPartitionResolver(resolver);
        RegionPtr regionPtr = regionFactory->create( "Trades" );
    }
```

Using AttributesMutator to Modify a Plug-In

A cache listener, cache loader or cache writer can be added to or removed from a region after the region is created by retrieving and running the `Region` object's `AttributesMutator`. Mutable attributes define operations that are run from the client itself.

This example shows how to use `AttributesMutator` to dynamically add a cache listener to an existing region.

Example 2.21 Dynamically Adding a Cache Listener to a Region

```
void setListener(RegionPtr& region)
{
  CacheListenerPtr regionListener = new TestCacheListener();
  AttributesMutatorPtr regionAttributesMutator =
       region->getAttributesMutator();

  // Change cache listener for region.
  regionAttributesMutator->setCacheListener(regionListener);
}
```

The plug-ins can also be implemented using a dynamically linked library. The class is not available to the application code in this case, so a `factory` method is required by the `set` function along with the name of the library.

This example shows how to use `AttributesMutator` along with the `setCacheListener` function to obtain a new cache listener object using the `factory` function provided by the library. Next, the listener is set for the region.

Example 2.22 Adding a Cache Listener Using a Dynamically Linked Library

```
void setListenerUsingFactory(RegionPtr& region)
{
  AttributesMutatorPtr regionAttributesMutator =
       region->getAttributesMutator();

  // Change cache listener for region.
  regionAttributesMutator->setCacheListener("Library",
"createTestCacheListener");
}
```

To use `AttributesMutator` to remove a plug-in from a region, set the plug-in's value to `NULLPTR`, as shown in the following example.

Example 2.23 Removing a Cache Listener From a Region

```
void removeListener(RegionPtr& region)
{
  CacheListenerPtr nullListener = NULLPTR;
  AttributesMutatorPtr regionAttributesMutator =
       region->getAttributesMutator();

  // Change cache listener for region to NULLPTR
  regionAttributesMutator->setCacheListener(nullListener);
}
```

Considerations for Implementing Callbacks

Keep your callback implementations lightweight and prevent situations that might cause them to hang. For example, do not perform distribution operations or disconnects inside events.

Your code should handle any exceptions that it generates. If not, GemFire handles them as well as possible. Because C++ has no standard for exceptions, in many cases GemFire can only print an `unknown error` message.

Expiration Attributes

Expiration attributes govern the automatic eviction of regions and region entries from the cache. Eviction is based on the time elapsed since the last update or access to the object. This is referred to as the least-recently-used (LRU) eviction process. Expiration options range from marking the expired object as invalid to completely removing it from the distributed cache. Eviction can help keep data current by removing outdated entries, prompting a reload the next time they are requested. Eviction may also be used to recover space in the cache by clearing out unaccessed entries and regions.

Similar to application plug-ins, expiration activities are hosted by each application that defines a region in its cache.

The following example shows a declaration that causes the region's entries to be invalidated in the local cache after they have not been accessed for one minute.

Example 2.24 Declaring Expiration Attributes

```
<region-attributes>
   <entry-idle-time>
      <expiration-attributes timeout="60" action="local-invalidate"/>
   </entry-idle-time>
</region-attributes>
```

Region and region entry expiration attributes are set at the region level. By default, regions and entries do not expire. The following attributes cover two types of expiration: time-to-live (TTL) and idle timeout.

`RegionTimeToLive`	The amount of time, in seconds, that the region remains in the cache after the last creation or update before the expiration action occurs.
`EntryTimeToLive`	For entries, the counter is set to zero for `create` and `put` operations. Region counters are reset when the region is created and when an entry has its counter reset. An update to an entry causes the time-to-live (TTL) counters to be reset for the entry and its region.
`RegionIdleTimeout`	The amount of time, in seconds, that the region remains in the cache after the last access before the expiration action occurs.
`EntryIdleTimeout`	The idle timeout counter for an object is reset when its TTL counter is reset. An entry's idle timeout counter is also reset whenever the entry is accessed through a `get` operation. The idle timeout counter for a region is reset whenever the idle timeout is reset for one of its entries.

Using Statistics to Measure Expiration

Expiration is measured by comparing expiration attribute settings with the last accessed time and last modified time statistics. These statistics are directly available to applications through the `CacheStatistics` object that is returned by the `Region::getStatistics` and `RegionEntry::getStatistics` functions. The `CacheStatistics` object also provides a function for resetting the statistics counters.

Expiration Actions

You can specify one of the following actions for each expiration attribute:

- **Destroy**—Remove the object completely from the cache. For regions, all entries are destroyed as well. Destroy actions are distributed according to the region's distribution settings.
- **Invalidate**—Mark the object as invalid. For entries, the value is set to `NULL`. Regions are invalidated by invalidating all of its entries. Invalidate actions are distributed according to the region's distribution settings. When an entry is invalid, a `get` causes the cache to retrieve the entry according to the steps described in *Entry Retrieval* on page 51.
- **Local destroy**—Destroy the object in the local cache but do not distribute the operation.
- **Local invalidate**—Invalidate the object in the local cache but do not distribute the operation.

Destruction and invalidation cause the same event notification activities whether carried out explicitly or through expiration activities.

Region Expiration

Expiration activities in distributed regions can be distributed or performed only in the local cache. So one cache could control region expiration for a number of caches in the distributed system.

2.6 Managing the Lifetime of a Cached Object

All cacheable objects derive from `SharedBase`, which provides reference counting. Cacheable objects are referenced using `SharedPtr` types.

A native client may have many pointers that reference an object. When `SharedPtr` retrieves a cached object, the object remains alive as long as that pointer or the cache itself references the object. Regardless of how many pointers to the object are deleted, the object remains alive until the last remaining pointer is deleted. At that point the object is deleted.

This is a very simple example:

```
CacheableStringPtr p = CacheableString::create ("string");
region.put ("key", p) ;
```

- In the example, the act of object creation allocates memory and initializes the object.
- When you assign the object to a `SharedPtr`, you relinquish control of the lifetime of that object to the reference counting mechanism for the cache.
- The `put` operation does not actually copy the object into the cache. Rather, it copies a `SharedPtr` into the cache's hashmap. Consequently, the object remains alive in the cache when the original `SharedPtr` goes away.

The client can make use of an object after you have initialized the object. For example, another `SharedPtr` might issue a `get` to retrieve the object from the cache:

```
CacheableStringPtr p2 = region.get ("key") ;
```

Because `p` (the original `SharedPtr`) and `p2` point to the same object in memory, it is possible under some circumstances for multiple `SharedPtr` types to work on the same object in data storage.

Once you have put an object into the cache, do not delete it explicitly. Attempting to do so can produce undesirable results.

Changed Objects

If an object update is received, the cache no longer holds the same object. Rather, it holds a completely different instance of the object. The client does not see the updates until it calls a `get` to fetch the object again from the local cache, or (in a cache plug-in) calls `EntryEvent::getNewValue`.

For more about plug-ins, see *Application Plug-Ins* on page 61.

Object Expiration

When a cache automatically deletes an object as a result of an expiration action, the reference counting pointers protect the client from situations that might otherwise result if the cache actually freed the object's memory. Instead, the client disconnects the object from the cache by deleting the cache's `SharedPtr` reference, while leaving untouched any client threads with a `SharedPtr` to that object.

Object Lifetime Across the Distributed Cache

An object remains alive until every copy of the object is gone. In distributed regions, expiration activities can be local or distributed, depending on a region's distribution settings. One cache could control the expiration of all copies of an object in all the caches in the distributed system. Alternatively, each cache could control the expiration of its own local copy of the object. If the configuration gives each cache local control, and the expiration parameters are set to different lengths of time in different caches, some copies of an object may still exist after it has disappeared in other caches. See *Expiration Attributes* on page 69 for more information.

2.7 Client to Server Connection Process

This functional overview describes the top-level sequence of native client connection events with a GemFire cache server:

1. A native client region is configured in `cache.xml` or programmatically with a set of server connection endpoints. Server endpoints identify each cache server by specifying the server's name and port number.

   Client threads obtain, use, and release a connection to a connection pool that maintains new connections. The number of connections per active endpoint (which is either per region, or cache-level for querying) is governed by the native client `connection-pool-size` system property in the `gfcpp.properties` file. The default value for `connection-pool-size` is `5`. If `connection-pool-size` is set to `0` and `client-notification` is set to `true`, the client only listens for events.

   This example shows how to use `cache.xml` to configure a native client region with endpoints set to two cache servers:

```
<pool name="examplePool" subscription-enabled="true" >
   <server host="java_servername1" port="java_port1" />
   <server host="java_servername2" port="java_port2" />
</pool>
<region name="NativeClientRegion" refid="CACHING_PROXY">
   <region-attributes pool-name="examplePool"/>
</region>
```

   TCP connections on the native client are specified at the cache level, or by overriding endpoints for specific regions. The connections are created as the regions are created. In addition, connections can also get created for querying without having any created regions. In this case, when endpoints are defined at the cache level no regions are yet created and a query is fired.

   You can configure client-server connections in two ways. Use either the region/cache endpoints or the Pool API. For more information about the pool API, see *Using Connection Pools* on page 235.

2. The client announces to the server which entries it wishes to have updated by programmatically registering interest in those entries. See *Registering Interest for Entries* on page 46 for more information.

3. The client `cache.xml` file should have the following parameters configured so the client can update the server and the client can receive updates from the server:

   - Caching enabled in the client region, by using the `CACHING_PROXY` `RegionShortcut` setting in the region attribute `refid`. A listener could also be defined so event notification occurs. You can use both, but at least one of the two methods must be used by the client to receive event notifications.
   - Set `client-notification` to `true` so the client receives update notifications from the server for entries to which it has registered interest.

4. A native client application calls the C++ or .NET API to connect to a cache server.

5. The client and the cache server exchange a handshake over a configured endpoint to create a connection.

6. Any `create`, `put`, `invalidate`, and `destroy` events sent to the server are propagated across the distributed cache so the client can receive the events.

   You may be able to improve system performance by making adjustments to the cache server. See the GemFire Enterprise System Administrator's Guide *for cache server tuning information.*

2.8 Troubleshooting

Cannot Acquire Windows Performance Data

When you attempt to run performance measurements for the native client on Windows, you may encounter the following error message in the run logs:

```
Can't get Windows performance data. RegQueryValueEx returned 5
```

This can occur because incorrect information is returned when a Win32 application calls the ANSI version of `RegQueryValueEx` Win32 API with `HKEY_PERFORMANCE_DATA`. This error is described in Microsoft KB article ID 226371 at http://support.microsoft.com/kb/226371/en-us.

To successfully acquire Windows performance data, you need to verify that you have the proper registry key access permissions in the system registry. In particular, make sure that `Perflib` in the following registry path is readable (`KEY_READ` access) by the GemFire process:

```
HKEY_LOCAL_MACHINE\
    SOFTWARE\
    Microsoft\
    Windows NT\
    CurrentVersion\
    Perflib
```

An example of reasonable security on the performance data would be to grant administrators `KEY_ALL_ACCESS` and interactive users `KEY_READ` access. This particular configuration prevents non-administrator remote users from querying performance data.

See http://support.microsoft.com/kb/310426 and http://support.microsoft.com/kb/146906 for instructions about how to ensure that GemFire processes have access to the registry keys associated with performance.

Generating a Process Memory Dump Image

You can generate a process memory dump image (core files in Unix systems and minidumps in Windows). The image is produced when a fatal error occurs that normally terminates the program.

When the system property `crash-dump-enabled` is set to `true`, a dump image is generated (the default is `true`). The dump file is generated in the same location as the `log-file` directory, and has the same prefix as the log file. The name is `<prefix>-<time>.core.<pid>` in Unix, and `<prefix>-<time>-<pid>.dmp` in Windows).

Unix systems generate core files automatically for such errors, but this option is useful for providing a custom location and name, as well as for systems where core dump generation is disabled. For Unix, when system core dump generation is turned on (`ulimit -c`) this property can be set to `false`.

For .NET clients, when this property is set then `AccessViolation` exceptions are trapped and a crash dump is created to assist with further analysis. Applications receive a `FatalInternalException` for this case, with the `InnerException` set to the originating `AccessViolationException`.

This requires the availability of `dbghelp.dll` on Windows, either in the same directory as `gfcppcache.dll` or in the system `PATH`. The file is installed by default, though for Windows 2000 a newer version may be required for minidumps. For Unix systems, the `gcore` command should be available (gdb > 5.2 on Linux; available by default in Solaris).

Chapter
3
Cache Initialization File

To ease the task of managing the structure of the cache, you can define the default cache structure in an XML-based initialization file. To modify the cache structure, you just edit `cache.xml` in your preferred text editor. No changes to the application code are required.

This chapter describes the file format of the `cache.xml` file and discusses its contents.

In this chapter:

3.1 Cache Initialization File Basics

The contents of the cache initialization file are used to populate or update a cache, either when a cache server starts up or when a client application explicitly creates its cache or explicitly loads a new structure into an existing cache. The initialization file can have any name, but is generally referred to as `cache.xml`. Both client applications and cache servers can use an optional `cache.xml` file to ease the initialization process.

File Contents

The contents of a declarative XML file correspond to APIs declared in the `Cache.hpp` and `Region.hpp` header files. The cache initialization file allows you to accomplish declaratively many of the cache management activities that you can program through the API.

- The contents of the cache initialization file must conform to the XML definition in *`productDir`*`/dtd/gfcpp-cache3500.dtd` (see *Native Client Cache XML DTD* on page 78).
- The name of the declarative XML file is specified when establishing a connection to the distributed system. You can define it by setting the `cache-xml-file` configuration attribute in the `gfcpp.properties` file for the native client. For details about the `gfcpp.properties` file, see *Setting Properties* on page 139.

3.2 Example cache.xml File

This section presents an example `cache.xml` file that shows cache and region initialization for a client. This example shows only a subset of the possible data configurations. For detailed information about cache and region configuration, including the default attribute settings, see Chapter 2, *The Native Client Cache*, on page 37. Also see the online API documentation for `Cache` and `RegionAttributes`.

For information a cache with server pool, see *Using Connection Pools* on page 235. The example below shows a `cache.xml` file that creates two regions.

- Region `region1` is defined with a full set of region attributes and application plug-ins. The region's entries have `RegionTimeToLive` and `RegionIdleTimeout` expiration attributes set (page 69).
- Region `region2` uses mostly default values.

Example 3.1 Declarative Cache Initialization with cache.xml

```
<<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 3.5//EN"
    "http://www.gemstone.com/dtd/gfcpp-cache3500.dtd">
<!-- Sample cache.xml file -->
<!-- Example 3.1 Declarative Cache Initialization with cache.xml -->
<cache>
   <pool name="examplePool" subscription-enabled="true">
       <server host="localhost" port="24680" />
   </pool>
   <region name="root1" refid="CACHING_PROXY">
     <region-attributes pool-name="examplePool"
       initial-capacity="25"
       load-factor="0.32"
       concurrency-level="10"
       lru-entries-limit="35">
       <region-idle-time>
         <expiration-attributes timeout="20" action="destroy"/>
       </region-idle-time>
       <entry-idle-time>
         <expiration-attributes timeout="10" action="invalidate"/>
       </entry-idle-time>
       <region-time-to-live>
         <expiration-attributes timeout="5" action="local-destroy"/>
       </region-time-to-live>
       <entry-time-to-live>
         <expiration-attributes timeout="10" action="local-invalidate"/>
       </entry-time-to-live>
     </region-attributes>
   </region>
</cache>
```

For details about the individual region attributes, see *Region Attributes* on page 53.

3.3 Native Client Cache XML DTD

The following example presents the file `gfcpp-cache3500.dtd`, which defines the XML used by the GemFire Enterprise native client for declarative caching. The contents of the cache initialization file must conform to the data type definitions in `gfcpp-cache3500.dtd`. The DTD file identifies the valid element tags that may be present in your XML file, the attributes that correspond to each element, and the valid values for the elements and attributes. In this example, everything that is not a comment is shown in **`bold`**.

The `gfcpp-cache3500.dtd` file can be found in the *`productDir`*`/dtd` directory of your native client installation.

Example 3.2 gfcpp-cache3500.dtd

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
This is the XML DTD for the GemFire Enterprise -C++ distributed cache declarative
caching XML file.
The contents of a declarative XML file correspond to APIs found in the
Gemfire Enterprise -C++ product, more specifically in the
Cache.hpp and Region.hpp files in the product include directory
A declarative caching XML file is used to populate a Cache
when it is created.
-->

<!--
The "cache" element is the root element of the declarative cache file.
This element configures a GemFire Enterprise -C++ Cache and describes the
root regions it contains, if any.
-->
<!ELEMENT cache (pool*, root-region*, region*)>
<!ATTLIST cache
  endpoints CDATA #IMPLIED
  redundancy-level CDATA #IMPLIED
>

<!--
A "locator" element specifies the host and port that a server locator is listening on
-->
<!ELEMENT locator EMPTY>
<!ATTLIST locator
  host  CDATA #REQUIRED
  port  CDATA #REQUIRED
>

<!--
A "server" element specifies the host and port that a cache server is listening on
-->
<!ELEMENT server EMPTY>
<!ATTLIST server
  host  CDATA #REQUIRED
  port  CDATA #REQUIRED
>

<!--
A "pool" element specifies a client-server connection pool.
-->
```

```
<!ELEMENT pool (locator+|server+)>
<!ATTLIST pool
  free-connection-timeout        CDATA #IMPLIED
  load-conditioning-interval     CDATA #IMPLIED
  min-connections                CDATA #IMPLIED
  max-connections                CDATA #IMPLIED
  retry-attempts                 CDATA #IMPLIED
  idle-timeout                   CDATA #IMPLIED
  ping-interval                  CDATA #IMPLIED
  name                           CDATA #REQUIRED
  read-timeout                   CDATA #IMPLIED
  server-group                   CDATA #IMPLIED
  socket-buffer-size             CDATA #IMPLIED
  subscription-enabled                  (false | true) #IMPLIED
  subscription-message-tracking-timeout CDATA #IMPLIED
  subscription-ack-interval             CDATA #IMPLIED
  subscription-redundancy        CDATA #IMPLIED
  statistic-interval             CDATA #IMPLIED
  pr-single-hop-enabled          (true | false) #IMPLIED
  thread-local-connections       (false | true) #IMPLIED
  multiuser-authentication          (false | true) #IMPLIED
>

<!--
A root-region" element describes a root region whose entries and
subregions will be stored in memory.
Note that the "name" attribute specifies the simple name of the region;
it cannot contain a "/".
-->
<!ELEMENT root-region (region-attributes?, region*)>
<!ATTLIST root-region
  name CDATA #REQUIRED
>

<!--
A "region" element describes a region (and its entries) in GemFire
Enterprise -C++ distributed cache.  It may be used to create a new region or may be
used to add new entries to an existing region.  Note that the "name"
attribute specifies the simple name of the region; it cannot contain a
"/".
-->
<!ELEMENT region (region-attributes?, region*)>
<!ATTLIST region
  name CDATA #REQUIRED
>

<!--
A "region-attributes" element describes the attributes of a region to
be created. For more details see the AttributesFactory header in the
product include directory
-->
<!ELEMENT region-attributes ((region-time-to-live |
  region-idle-time | entry-time-to-live | entry-idle-time |
  partition-resolver |
  cache-loader | cache-listener | cache-writer | persistence-manager)*)>
<!ATTLIST region-attributes
  caching-enabled (true | TRUE | false | FALSE) #IMPLIED
  cloning-enabled (true | TRUE | false | FALSE) #IMPLIED
  scope (local | distributed-no-ack | distributed-ack ) #IMPLIED
  initial-capacity CDATA #IMPLIED
  load-factor CDATA #IMPLIED
  concurrency-level CDATA #IMPLIED
  lru-entries-limit CDATA #IMPLIED
  disk-policy (none | overflows | persist ) #IMPLIED
  endpoints CDATA #IMPLIED
  client-notification (true | TRUE | false | FALSE) #IMPLIED
  pool-name CDATA #IMPLIED
  id CDATA #IMPLIED
  refid CDATA #IMPLIED
>

<!--
A "region-time-to-live" element specifies a Region's time to live
-->
<!ELEMENT region-time-to-live (expiration-attributes)>

<!--
A "region-idle-time" element specifies a Region's idle time
-->
<!ELEMENT region-idle-time (expiration-attributes)>

<!--
A "entry-time-to-live" element specifies a Region's entries' time to
live
-->
<!ELEMENT entry-time-to-live (expiration-attributes)>

<!--
A "entry-idle-time" element specifies a Region's entries' idle time
-->
<!ELEMENT entry-idle-time (expiration-attributes)>

<!--
A "properties" element specifies a persistence properties
-->
<!ELEMENT properties (property*)>

<!--
An "expiration-attributes" element describes expiration
-->
<!ELEMENT expiration-attributes EMPTY>
<!ATTLIST expiration-attributes
  timeout CDATA #REQUIRED
  action (invalidate | destroy | local-invalidate | local-destroy) #IMPLIED
>

<!--
A "cache-loader" element describes a region's CacheLoader
-->
<!ELEMENT cache-loader  EMPTY >
<!ATTLIST cache-loader
 library-name CDATA #IMPLIED
 library-function-name CDATA #REQUIRED
>

<!--
A "cache-listener" element describes a region's CacheListener
-->
```

```
<!ELEMENT cache-listener EMPTY>
<!ATTLIST cache-listener
 library-name CDATA #IMPLIED
 library-function-name CDATA #REQUIRED
>

<!--
A "cache-writer" element describes a region's CacheListener
-->
<!ELEMENT cache-writer EMPTY>
<!ATTLIST cache-writer
 library-name CDATA #IMPLIED
 library-function-name CDATA #REQUIRED
>

<!--
A "partition-resolver" element describes a region's PartitionResolver
-->
<!ELEMENT partition-resolver EMPTY>
<!ATTLIST partition-resolver
 library-name CDATA #IMPLIED
 library-function-name CDATA #REQUIRED
>

<!--
A "persistence-manager" element describes a region's persistence feature
-->
<!ELEMENT persistence-manager (properties)>
<!ATTLIST persistence-manager
 library-name CDATA #IMPLIED
 library-function-name CDATA #REQUIRED
>

<!--
A "property" element describes a persistence property
-->
<!ELEMENT property EMPTY>
<!ATTLIST property
 name CDATA #REQUIRED
 value CDATA #REQUIRED
>
```

Chapter 4

The C++ Caching API

The native client C++ caching API allows C++ clients to manage data in a GemFire Enterprise system. The online C++ API documentation (included in the `docs/cppdocs` directory of the GemFire Enterprise native client product installation) provides extensive implementation details for the C++ structures and functions.

Several example API programs are included in the *`productDir`*`/examples` directory. See *Running the Product Examples and QuickStart Guide* on page 36.

In this chapter:

4.1 The GemFire C++ API

The GemFire C++ API allows C++ and .NET developers to programmatically create, populate, and manage a GemFire distributed system. The C++ library is thread-safe, except where specified otherwise in the online API documentation.

This section gives a general overview of the classes in the `gemfire`, `gemfire_statistics`, and `gemfire_admin` namespaces. For complete and current information on the classes listed here, see the online C++ API documentation.

Cache

- **`CacheFactory`**—This class creates a `Cache` instance based on an instance of `DistributedSystem`. If `cache.xml` is specified by `DistributedSystem`, the cache is created based on the declarations loaded from that file.
- **`Cache`**—This is the entry point to the client caching API. You create regions.with this class. The cache is created by calling the `create` function of the factory class, `CacheFactory`. When creating a cache, you specify a `DistributedSystem` that tells the new cache where to find other caches on the network and how to communicate with them.

Region

- **`Region`**—This class provides functions for managing regions and cached data. The functions for this class allow you to perform the following actions:
  - Retrieve information about the region, such as its parent region and region attribute objects.
  - Invalidate or destroy the region.
  - Create, update, invalidate and destroy region entries.
  - Retrieve region entry keys, entry values, and `RegionEntry` objects, either individually or as entire sets.
  - Retrieve the `statistics` object associated with the region.
  - `Set` and `get` user-defined attributes.
- **`RegionEntry`**—This class contains the key and value for the entry, and provides all non-distributed entry operations. This object's operations are not distributed and do not affect statistics.

Region Attributes

- **`RegionAttributes`**—This class holds all attribute values for a region and provides functions for retrieving all attribute settings. This class can be modified by the `AttributesMutator` class after region creation.
- **`AttributesMutator`**—This class allows modification of an existing region's attributes for application plug-ins and expiration actions. Each region has an `AttributesMutator` instance.

Application Plug-Ins

- **`CacheLoader`**—Application plug-in class for loading data into a region on a cache miss.
- **`CacheWriter`**—Application plug-in class for synchronously handling region and entry events before the events occur. Entry events are `create`, `update`, `invalidate`, and `destroy`. Region events are `invalidate` and `destroy`. This class has the ability to abort events.
- **`CacheListener`—**Application plug-in class for handling region and entry events after they occur. Entry events are `create`, `update`, `invalidate`, and `destroy`. Region events are `invalidate` and `destroy`.

Event Handling

- **`RegionEvent`**—This class provides information about the event, such as what region the event originated in, whether the event originated in a cache remote to the event handler, and whether the event resulted from a distributed operation.
- **`EntryEvent`**—This class provides all of the information available for the `RegionEvent`, and provides entry-specific information such as the old and new entry values and whether the event resulted from a `load` operation.

Statistics API

The `StatisticsType` API represents a blueprint for the same type of `Statistics`. The `StatisticsType` API is a collection of `StatisticDescriptor`. Internally, each `StatisticDescriptor` describes data of each individual statistic. `StatisticsFactory` provides functionality for creating `StatisticDescriptor`, `StatisticsType`, and `Statistics`.

- **`CacheStatistics`**—This class defines common statistics functions. `Region` and `RegionEntry` both have functions that return a `CacheStatistics` object for accessing and resetting their statistics counts.
- **`StatisticDescriptor`**—An instance of this class describes a statistic whose value is updated by an application and may be archived by the native client. Each statistic has a type of either `int`, `long`, or `double`, and either a gauge or a counter. The value of a gauge can increase and decrease, and the value of a counter strictly increases. The `StatisticDescriptor` is created by calling one of these `StatisticsFactory` functions: `createDoubleCounter`, `createDoubleGauge`, `createIntCounter`, `createIntGauge`, `createLongCounter`, `createLongGauge`.
- **`StatisticsType`**—An instance of this class describes a logical collection of `StatisticDescriptors`. These descriptions are used to create an instance of `Statistics`. The `StatisticsType` is created by calling `StatisticsFactory::createType`.
- **`Statistics`**—An instance of this class represents concrete `Statistics` of the associated `StatisticsType`. This class stores data related to all individual statistic objects. You can create an instance by calling `StatisticsFactory::createStatistics`. This class has functions to get, set, and increment statistic values.
- **`StatisticsFactory`**—This class provides functions for creating instances of `StatisticDescriptor`, `StatisticsType`, and `Statistics` objects. This is a singleton class, and its instance can be acquired using `StatisticsFactory::getExistingInstance`.

If you need to create new statistics, see *Creating New Statistics* on page 104 for details.

4.2 Creating a Cache

The code snippets in this section show cache creation. When you create your cache, the system automatically connects your process to the server tier.

For systems with security enabled, the credentials for a connecting client are authenticated when it creates the cache. See *Security* on page 163 for more information about authenticated connections.

In this example, the application creates the cache by calling the `CacheFactory::create` function, specifying the servers to connect to.

Example 4.1 Creating the System Connection and the Cache

```
CacheFactoryPtr cacheFactory = CacheFactory::createCacheFactory();
CachePtr cachePtr = cacheFactory
      ->addServer("localhost", 40404)
      ->addServer("localhost", 40405)
      ->setSubscriptionEnabled(true)
      ->create();
```

4.3 Cache Eviction and Overflow

You can manage the amount of memory a region consumes through data entry and region expiration settings, and with least recently used (LRU) eviction settings. Entry expiration allows you to remove data that has not been updated or accessed for a specified amount of time. See *Expiration Attributes* on page 69 for more information about entry expiration.

When eviction is configured, memory consumption or entry count is monitored and, when capacity is reached, reduces memory usage or region growth by removing or overflowing to disk the stalest LRU entries.

If you use disk data overflow to supplement memory for your data cache, you must make sure you have enough disk space to store the data. To overflow data, you specify an eviction capacity limit and set overflow to disk as the method used for the eviction action. Whenever the limit is reached, the least recently used data is written to disk and removed from the cache.

Creating a Region With Caching and LRU

The following example shows how to create a region with caching and LRU enabled. The application overrides the default region attributes where necessary. Note the operations of these functions in the code sample:

- The `setLruEntriesLimit` function sets the maximum number of entries to hold in a caching region to `20000`.
- The `setInitialCapacity` function sets the number of entries a caching region should initially expect to hold to `20000`.

For all other region attributes, the defaults are used.

Example 4.2 Creating a Region With Caching and LRU

```
RegionFactoryPtr regionFactory =
       cachePtr->createRegionFactory(CACHING_PROXY);
regionPtr = regionFactory->setLruEntriesLimit( 20000 )
   ->setInitialCapacity( 20000 )
   ->create("exampleRegion");
```

Handling Cache LRU

By default, the client cache is kept in memory. When the value for `LruEntriesLimit` is exceeded, those LRU entries are destroyed. See *LruEntriesLimit* on page 57 for details. With cache overflow, over-limit LRU entry values are written to disk, keeping the key in memory. When one of those entries is requested, its value is loaded back into memory.

The disk policy specifies whether to destroy or write overflow entries. See *DiskPolicy* on page 57.

A persistence manager API is provided for customers who need to provide their own mechanism for writing to disk or to a backend database. The provided persistence manager uses the open source Berkeley DB library. See *PersistenceManager* on page 57 for a description of Berkeley DB. The following example shows how to programmatically set up these attributes during region creation.

Example 4.3 Creating a Region With Disk Overflow Based on Entry Capacity

```
/** set up an empty client region */
   RegionFactoryPtr regionFactory = cachePtr->createRegionFactory(CACHING_PROXY);

   PropertiesPtr bdbProperties = Properties::create();
   bdbProperties->insert("PersistenceDirectory", "BDB");
   bdbProperties->insert("EnvironmentDirectory", "BDBEnv");
   //Use either of the two for setting the size
   bdbProperties->insert("CacheSizeGb", "1");
   //or use this
   //bdbProperties->insert("CacheSizeMb","512");
   bdbProperties->insert("PageSize", "65536");
   bdbProperties->insert("MaxFileSize","512000");

   RegionPtr regionPtr = regionFactory
      ->setLruEntriesLimit( 20000 )
      ->setInitialCapacity( 20000 )
      ->setDiskPolicy( DiskPolicyType::OVERFLOWS )
      ->setPersistenceManager("BDBImpl","createBDBInstance",bdbProperties)
      ->create("exampleRegion");
```

Cache Memory Size and Eviction

The `heap-lru-limit` property in `gfcpp.properties` sets the maximum amount of memory used for a cache, in megabytes. If a new entry causes memory to grow past this limit, the least-recently-used (LRU) algorithm is called to evict entries until memory size falls below the limit. See `heap-lru-limit` (page 145) for additional information.

The related `heap-lru-delta` property is the extra percentage of the memory above the heap limit over and above the actual percentage (`heap-lru-limit` plus `heap-lru-delta` percent) that needs to be evicted to bring the heap size back to the limit. Memory is reclaimed until the usage is below `heap-lru-limit` minus `heap-lru-delta` percent. See `heap-lru-delta` (page 145) for more information.

Heap LRU causes eviction to occur on all regions in the cache, effectively overriding region-level LRU settings when it needs to reclaim memory. The following points should be considered when you configure LRU eviction:

- All regions in the cache should have a persistence manager configured. See *PersistenceManager* on page 57 for more information. If there is no persistence manager, overlimit entries are destroyed.
- When `heap-lru-limit` is specified then LRU is enabled, regardless of whether `LruEntriesLimit` (page 57) is specified.
- `heap-lru-limit` applies to all regions in the cache. When the heap grows past this limit, entries are removed from one or more regions as necessary.

  Evicted entries can be saved to disk as an extension of the cache, rather than destroyed. See DiskPolicy on page 57 *for details.*

4.4 Adding an Entry to the Cache

A native client region can be populated with cache entries using the `Region::put` or the `Region::create` API functions. The `put` function places a new value into a region entry with the specified key, while the `create` function creates a new entry in the region. Both functions provide a user-defined parameter object to any cache writer invoked in the process, and new values for both functions are propagated to a connected cache server.

When the `put` function adds an entry, the previous value is overwritten if there is already an entry associated with the specified key in the region. In the next example, the program uses the API to `put` 100 entries into the cache by iteratively creating keys and values, both of which are integers.

Example 4.4 Using the API to Put Entries Into the Cache

```
for ( int32_t i=0; i < 100; i++ ) {
   regionPtr->put( i, CacheableInt32::create(i) );
}
```

Bulk Put Operations Using putAll

You can batch up multiple key/value pairs into a hashmap and put them into the cache with a single operation using the `Region::putAll` API function (`Region.PutAll` for .NET). Each entry is processed for interest registration on the server, so each entry requires its own unique event ID. Updates and creates can be mixed in a `putAll` operation, so those events need to be addressed on the cache server for appropriate cache listener invocation on distributed system members. Map entries retain their original order when they are processed at the server, The following table lists the client and cache server statistics for `putAll`.

Table 4.1 PutAll Statistics for Cache Server and Client

Statistic Type	Chart Name	Description
`CachePerfStats`	`Putalls`	Total number of times a map is added or replaced in the cache as a result of a local operation. Also reports the number of `putAll` operations.
`CacheperfStats`	`putallTime`	Total time to replace a map in the cache as a result of a local operation.
`CacheServerStats`	`putAllRequests`	Number of `putAll` requests.
`CacheServerStats`	`putAllResponses`	Number of `putAll` responses written to the cache client.
`CacheServerStats`	`processPutAllTime`	Total time to process a cache client `putAll` request, including the time to put all objects into the cache.
`CacheServerStats`	`readPutAllRequestTime`	Total time to read `putAll` requests.
`CacheServerStats`	`writePutAllResponseTime`	Total time to write putAll responses.
`CacheClientStats`	`putAll`	Number of `putAll` requests sent to the cache server.
`CacheClientStats`	`sendPutAllTime`	Total time for `sendPutAll`.

4.5 Accessing an Entry

The standard `Region::get` API method returns the value associated with the specified key, and passes the callback argument to any cache loaders or cache writers that are invoked in the operation. If the value is not present locally then it is requested from the cache server. If the cache server request is unsuccessful then a local cache loader is invoked.

The entry value is either retrieved from the local cache or fetched by the region's locally defined cache loader.

In the following example, the program uses the API to do a `get` for each entry that was put into the cache in Example 4.4 on page 91.

Example 4.5 Using the get API to Retrieve Values From the Cache

```
for ( int32_t i=0; i< 100; i++) {
   CacheableInt32Ptr res = dynCast<CacheableInt32Ptr>(regionPtr->get(i));
}
```

Bulk Get Operations Using getAll

You can use the `Region::getAll` API (`Region.GetAll` for .NET) to gets values for an array of keys from the local cache or server. If the value for a key is not present locally, then it is requested from the server.

> *The value returned is not copied, so multi-threaded applications should not modify the value directly, but should instead use the* `update` *methods.*

See the `Region` online API documentation for more information about using `getAll`.

4.6 Serialization

The native client provides a `Serializable` interface that you can use for fast and compact data serialization. This section discusses serialization, and presents a simple implementation example.

Built-In Types

The following table describes the set of built-in serializable types that are automatically registered at initialization.

Table 4.2 Built-In Cacheable Types Automatically Registered at Initialization

Cacheable Type	Description
`CacheableString`	Holds both `char*` strings and wide-character `wchar_t*` strings. The maximum length of the packed string is 64 kilobytes. An exception is thrown if this size limit is exceeded.
`CacheableByte`	For single byte values.
`CacheableBytes`	For multiple byte values.
`CacheableBoolean`	For bool values.
`CacheableDouble`	For double values.
`CacheableFloat`	For float values.
`CacheableInt16`	For 16-bit short integers.
`CacheableInt32`	For 32-bit integers.
`CacheableInt64`	For 64-bit long integers.
`CacheableWideChar`	For `wchar_t` wide characters.
`CacheableDoubleArray`	For arrays of double values.
`CacheableFloatArray`	For arrays of float values.
`CacheableInt16Array`	For arrays of 16-bit short integers.
`CacheableInt32Array`	For arrays of 32-bit integers.
`CacheableInt64Array`	For arrays of 64-bit long integers.
`CacheableStringArray`	For arrays of `CacheableString` values.
`CacheableVector`	For a `VectorT<>` of `CacheablePtr`.
`CacheableHashMap`	For a `HashMapT<>` of `CacheableKeyPtr` (key in the map) to `CacheablePtr` (value in the map).
`CacheableHashSet`	For a `HashSetT<>` of `CacheableKeyPtr` (hashset must have keys for a uniqueness check and for calculating hashcodes).
`CacheableObjectArray`	For arrays of `CacheablePtr`. This is actually a vector similar to `CacheableVector`, except for the `typeId` and `toData` or `fromData` which match the corresponding Java ones.

HashMapT<> and HashSetT<> Template Classes

Both `HashMapT<>` and `HashSetT<>` template classes require that the `hashcode` function and `equality` operator be defined for the key type. Additionally, the key type and value type for `HashMapT<>` should derive from `SharedBase`.

To implement a `hashcode` function and `equality` operator for a new type, overload the global functions as shown in the following example for new type `TKey`:

```
size_t gemfire::hashFunction(const TKey& k), and
bool gemfire::equalToFunction(const TKey& x, const TVal& y)
```

Alternatively, you can implement a `hashcode` function and == operator in class `TKey` (for the hashcode and equality operator, respectively), since that is the default implementation of the `hashFunction` and `equalToFunction` methods.

Complex Types

If your application uses more complex key types that you want to make more accessible or easier to handle, you can derive a new class from `CacheableKey`. For details, see *Custom Key Types* on page 98.

Another option is for the application to do its own object serialization using the GemFire `CacheableBytes` type or a custom type. See *Handling Data as a Blob* on page 101.

The GemFire `Serializable` *interface does not support object graphs with multiple references to the same object. If your application uses such circular graphs, you must address this design concern explicitly.*

How Serialization Works

When your application puts an object into the cache for subsequent distribution, GemFire serializes the data by taking these steps:

1. Calls the appropriate `classId` function.
2. Writes the full `typeId` using the `classId` for the instance.
3. Invokes the instance's `toData` function.

When your application subsequently receives a byte array, GemFire takes the following steps:

1. Decodes the `typeId`, extracts the `classId` from the `typeId`, then creates an object of the designated type using the registered factory functions.
2. Invokes the `fromData` function with input from the data stream.
3. Decodes the data, then populates the data fields.

Implementing the Serializable Interface

To store your own data types in the cache, you need to derive a new subclass from the `Serializable` interface. In practical terms, this means that you need to implement a small set of helper functions:

- Write a `toData` function that serializes your data.

  ```
  void toData (DataOutput& output)
  ```

  The `toData` function is responsible for copying all of the object's data fields to the object stream.

  The `DataOutput` class represents the output stream and provides methods for writing the primitives in a network byte order. For more about this, see the online API documentation for `DataOutput`.

- Write a `fromData` function that consumes a data input stream and repopulates the object's data fields.

  ```
  void fromData (DataInput& input)
  ```

  The `DataInput` class represents the input stream and provides methods for reading input elements. The `fromData` function must read the elements of the input stream in the same order that they were written by `toData`. For more about this, see the online API documentation for `DataInput`.

This example demonstrates a simple `BankAccount` class that encapsulates two `ints`: `ownerId` and `accountId`:

Example 4.6 The Simple Class BankAccount

```
class BankAccount
{
   private:

   int m_ownerId;
   int m_accountId;

   public:

   BankAccount( int owner, int account )
   : m_ownerId( owner ),
     m_accountId( account )
   {
   }

   int getOwner( )
   {
     return m_ownerId;
   }

   int getAccount( )
   {
     return m_accountId;
   }

};
```

To make `BankAccount` serializable, you would need to derive the class from `Serializable` and implement the following:

- **`toData`**—a function to serialize the data.
- **`fromData`**—a function to deserialize the data.
- **`classId`**—a function to provide a unique integer for the class.
- **`TypeFactoryMethod`**—a pointer to a function that returns a `Serializable*` to an uninitialized instance of the type.

The next example shows a code sample that demonstrates how to implement a serializable class.

Example 4.7 Implementing a Serializable Class

```
class BankAccount
: public Serializable
{
   private:
   int m_ownerId;
   int m_accountId;
   public:
   BankAccount( int owner, int account )
   : m_ownerId( owner ),
     m_accountId( account )
   {
   }
   int getOwner( )
   {
     return m_ownerId;
   }
   int getAccount( )
   {
     return m_accountId;
   }
   // Add the following for the Serializable interface
   // Our TypeFactoryMethod
   static Serializable* createInstance( )
   {
     return new BankAccount( 0, 0 );
   }
   int32_t classId( )
   {
     return 10; // must be unique per class.
   }
    virtual uint32_t objectSize() const
    {
       return 10;
    }
    void toData( DataOutput& output )
   {
     output.writeInt( m_ownerId );
     output.writeInt( m_accountId );
   }
   Serializable* fromData( DataInput& input )
   {
     input.readInt( &m_ownerId );
     input.readInt( &m_accountId );
     return this;
   }

};
```

Registering the Type

To be able to use the `BankAccount` type, you must register it with the type system so that when an incoming stream contains a `BankAccount`, it can be manufactured from the associated `TypeFactoryMethod`.

```
Serializable::registerType( BankAccount::createInstance );
```

Typically, you would register the type before calling the function `DistributedSystem::connect`.

Type IDs must be unique to only one class.

Custom Key Types

If your application uses key types that are too complex to easily force into `CacheableString`, you can likely improve performance by deriving a new class from `CacheableKey`. If you have hybrid data types you can implement your own derivation of `CacheableKey` that encapsulates the data type.

See *Serialization in Native Client Mode with a Java Server* on page 99 for information about implementing key types for a native client that is used with a Java cache server.

To extend a `Serializable` to be a `CacheableKey`, you need to modify the class definition as follows:

- Change the class so that it derives from `CacheableKey` rather than `Serializable`.
- Implement `operator==` and `hashcode` functions.

See the following example for sample code showing how to extend a serializable class to be a cacheable key.

Example 4.8 Extending a Serializable Class To Be a CacheableKey

```
class BankAccount
: public CacheableKey
{
   private:
   int m_ownerId;
   int m_accountId;
   public:
   BankAccount( int owner, int account )
   : m_ownerId( owner ),
     m_accountId( account )
   {
   }
   int getOwner( )
   {
     return m_ownerId;
   }
   int getAccount( )
   {
     return m_accountId;
   }
   // Our TypeFactoryMethod
   static Serializable* createInstance( )
   {
     return new BankAccount( 0, 0 );
   }
   int32_t typeId( )
   {
     return 1000; // must be unique per class.
   }
```

```
            void toData( DataOutput& output )
            {
              output.writeInt( m_ownerId );
              output.writeInt( m_accountId );
            }
            Serializable* fromData( DataInput& input )
            {
              input.readInt( &m_ownerId );
              input.readInt( &m_accountId );
              return this;
            }
            // Add the following for the CacheableKey interface
            bool operator == ( const CacheableKey& other ) const
            {
              const BankAccount& otherBA =
                static_cast<const BankAccount&>( other );
              return (m_ownerId == otherBA.m_ownerId) &&
                     (m_accountId == otherBA.m_accountId);
            }
            uint32_t hashcode( ) const
            {
              return m_ownerId;
            }
         virtual int32_t classId( )const
           {
             return 10; // must be unique per class.
           }

           virtual uint32_t objectSize() const
           {
             return 10;
           }
         };
```

Serialization in Native Client Mode with a Java Server

Primitive object types supported in all languages (`CacheableInt32`, `CacheableString`, `CacheableBytes`) function without requiring custom definitions with the Java cache server. For the keys, the Java cache server has to deserialize them and locate the hashcode to be able to insert the internal maps. Because of this, key types for C++ and .NET native clients used with a Java server are required to be registered on the Java server, but the value types do not need to be registered. This needs to be done even if there are no Java clients. The Java serializable type should have the same `classId` as the .NET class, and it should serialize and deserialize the type in the same manner as the .NET implementation.

See *Serialization* on page 120 for more information about .NET data serialization.

Implementing User-Defined Objects in Java Clients

There are two methods currently available for implementing a user-defined object: in a Java client that works with C++ clients: `Instantiator.register` and `DataSerializable`.

Instantiator.register

With the `Instantiator.register` method, a bridge client sends a `RegistrationMessage` to every Java VM in its distributed system. The message announces the mapping between a user-defined `classId` and class name. The other Java VMs can deserialize the byte array with the correct class.

If two bridge clients are in different distributed systems, then a `RegistrationMessage` cannot be sent to each other. For example: a `put` made by a client in one distributed system will hang when a client in another distributed system performs a `get` in pure Java mode. Similarly, a `put` made by a C++ client will cause a Java client to hang.

DataSerializable

Using the `DataSerializable` method, the user-defined object is serialized into the following byte array:

```
45 <2-byte-length> <class-name>
```

Another Java client in a different distributed system can deserialize the byte array, but a C++ client cannot convert the Java class name to a C++ class name.

Implementation

The `DataSerializable` method does not support using a nested object, while `Instantiator.register` does support the use of nested objects. A workaround is to let each Java client manually initiate an object for each possible user object class a C++ client provides, using the following code:

```
User u = new User("", 0);
```

See Example 14.7 on page 283 for a code sample that shows how to set up user object classes in a Java client.

Handling Object Graphs

If you have a graph of objects where each node can be serializable, the parent node can call `DataOutput::writeObject` to delegate the serialization responsibility to its child nodes. Similarly, your application can call `DataInput::readObject` to deserialize the object graph.

For more information, see the online API documentation for `DataOutput` and `DataInput`.

Handling Data as a Blob

If you have data that is best handled as a blob, such as structs that do not contain pointers, use the serializable type `CacheableBytes`. `CacheableBytes` is a blob class that implements the serialization for you.

`CacheableBytes` also provides direct access to the blob data. Because it is not derived from the `CacheableKey` interface, `CacheableBytes` enables you to modify data in place and then put it into the region again to distribute the change.

For more information, see the online API documentation for `CacheableB`ytes.

4.7 Using a Custom Class

This example shows how to use the defined `BankAccount` custom key type and the `AccountHistory` value type. The following example takes you through these basic operations: registering, creating a cache, connecting to the distributed system, putting data, getting data, and closing the cache.

Example 4.9 Using a BankAccount Object

```
#include <gfcpp/GemfireCppCache.hpp>
#include "BankAccount.hpp"
#include "AccountHistory.hpp"
using namespace gemfire;
/*
  This example connects, registers types, creates the cache, creates a
  region, and then puts and gets user defined type BankAccount.
*/
int main( int argc, char** argv )
{
// Register the user-defined serializable type.
    Serializable::registerType( AccountHistory::createDeserializable );
    Serializable::registerType( BankAccount::createDeserializable );

    CacheFactoryPtr cacheFactory = CacheFactory::createCacheFactory();
    // Create a cache.
    CachePtr cachePtr = cacheFactory->setSubscriptionEnabled(true)
        ->addServer("localhost", 24680)
        ->create();

    // Create a region.
    RegionFactoryPtr regionFactory =
        cachePtr->createRegionFactory(CACHING_PROXY);
    RegionPtr  regionPtr = regionFactory->create("BankAccounts");

// Place some instances of BankAccount cache region.
    BankAccountPtr KeyPtr(new BankAccount(2309, 123091));
    AccountHistoryPtr ValPtr(new AccountHistory());
    ValPtr->addLog( "Created account" );
    regionPtr->put( KeyPtr, ValPtr );
    printf( "Put an AccountHistory in cache keyed with BankAccount.\n" );
// Call custom behavior on instance of BankAccount.
    KeyPtr->showAccountIdentifier();
// Call custom behavior on instance of AccountHistory.
    ValPtr->showAccountHistory();
// Get a value out of the region.
    AccountHistoryPtr historyPtr =
        dynCast<AccountHistoryPtr>( regionPtr->get( KeyPtr ) );
    if ( historyPtr != NULLPTR ) {
        printf( "Found AccountHistory in the cache.\n" );
        historyPtr->showAccountHistory();
        historyPtr->addLog( "debit $1,000,000." );
        regionPtr->put( KeyPtr, historyPtr );
        printf( "Updated AccountHistory in the cache.\n" );
    }
// Look up the history again.
   historyPtr = dynCast<AccountHistoryPtr>( regionPtr->get( KeyPtr ) );
   if ( historyPtr != NULLPTR ) {
      printf( "Found AccountHistory in the cache.\n" );
      historyPtr->showAccountHistory();
   }
// Close the cache and disconnect from the servers
   cachePtr->close();
   return 0;
}
```

4.8 Creating New Statistics

The following example provides a programmatic code sample for creating and registering new statistics. For information about the `gemfire_statistics` API, see *Statistics API* on page 86.

Example 4.10 Creating New Statistics Programmatically

```
//Get StatisticsFactory
   StatisticsFactory* factory = StatisticsFactory::getExistingInstance();


//Define each StatisticDescriptor and put each in an array
   StatisticDescriptor** statDescriptorArr = new StatisticDescriptor*[6];
   statDescriptorArr[0] = statFactory->createIntCounter("IntCounter",
      "Test Statistic Descriptor Int Counter.","TestUnit");

   statDescriptorArr[1] = statFactory->createIntGauge("IntGauge",
      "Test Statistic Descriptor Int Gauge.","TestUnit");

   statDescriptorArr[2] = statFactory->createLongCounter("LongCounter",
      "Test Statistic Descriptor Long Counter.","TestUnit");

   statDescriptorArr[3] = statFactory->createLongGauge("LongGauge",
      "Test Statistic Descriptor Long Gauge.","TestUnit");

   statDescriptorArr[4] = statFactory->createDoubleCounter("DoubleCounter",
      "Test Statistic Descriptor Double Counter.","TestUnit");

   statDescriptorArr[5] = statFactory->createDoubleGauge("DoubleGauge",
      "Test Statistic Descriptor Double Gauge.","TestUnit");


//Create a StatisticsType
   StatisticsType* statsType = statFactory->createType("TestStatsType",
      "Statistics for Unit Test.",statDescriptorArr, 6);

//Create Statistics of a given type
   Statistics* testStat  =
      factory->createStatistics(statsType,"TestStatistics");


//Statistics are created and registered. Set and increment individual values
   Int statIdIntCounter = statsType->nameToId("IntCounter");
   testStat->setInt(statIdIntCounter, 10 );
   testStat->incInt(statIdIntCounter, 1 );
   int currentValue = testStat->getInt(statIdIntCounter);
```

Chapter 5

The .NET Caching API

The Microsoft .NET Framework interface for the GemFire Enterprise native client provides complete access to the native client C++ functionality from any .NET Framework language (C#, C++/CLI, VB.NET, J#). This enables C# clients, as well as those using other .NET languages, to use the capabilities provided by the C++ API.

The GemFire Enterprise native client calls a C++/CLI (Common Language Infrastructure) managed set of assemblies. C++/CLI includes the libraries and objects necessary for common language types, and it is the framework for .NET applications. C# is used as the reference language in this chapter, although any other .NET language works the same.

This chapter gives an overview of the functions of the caching API for .NET, and provides programming examples of their use. It also describes how to use the .NET Framework interface with the native client. The API documentation in the native client product `docs` directory provides extensive implementation details for the C++ and .NET structures and functions. Example API programs are included in the *`productDir`*`/examples` directory.

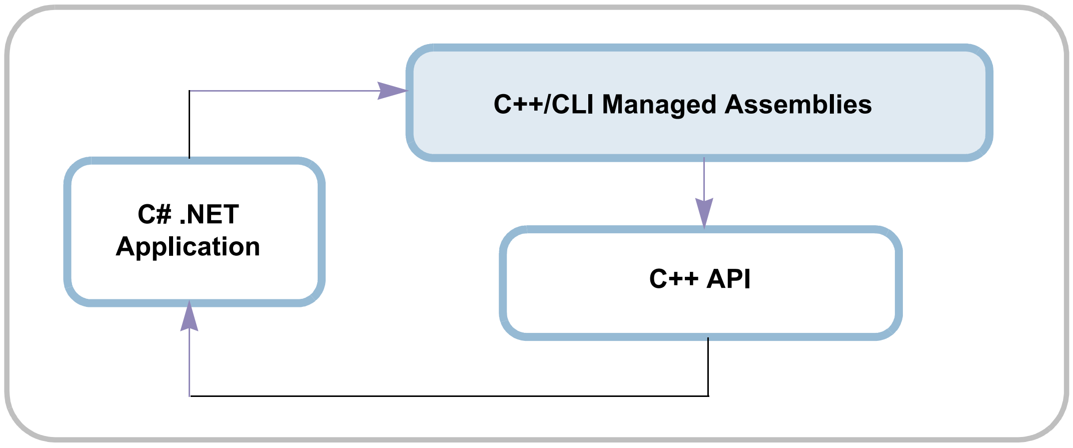
In this chapter:

5.1 Overview of the C# .NET API

The .NET API for the native client adds Microsoft .NET Framework CLI language binding for the GemFire Enterprise native client product.

The C# .NET API provides a CLI set of assemblies for the C++ API. Using C#, you can write callbacks and define user objects implementing the `serializable` interface. The following figure shows an overview of how a C# application accesses the native client C++ API functionality through C++/CLI.

Figure 5.1 C# .NET Application Accessing the C++ API



5.2 The GemFire C# .NET API

The GemFire C# .NET API allows you to programmatically create, populate, and manage a GemFire distributed system.

The C# .NET library is thread-safe, unless specified otherwise in the online .NET API documentation.

This section gives a general overview of the classes in the `GemStone::GemFire::Cache` namespace. For the complete and current information on the classes listed, see the online .NET API documentation.

C# .NET Naming Conventions

The following is a list of the .NET naming conventions used with the GemFire Enterprise native client:

- Unless noted, the .NET API classes and functions have the same names as their C++ counterparts in the namespace `GemStone::GemFire::Cache`. In C#, all method names start with a capital letter.
- The interfaces (`Serializable`, `CacheLoader`, `CacheListener`, and `CacheWriter`) have names starting with the letter `I`.
- The name of the `Serializable` interface is `IGFSerializable` because `ISerializable` is a .NET built-in type.
- Where possible, the `get`* and `set`* functions are replaced by .NET properties.
- You cannot extend any of the classes that are provided because they are marked as sealed; you can only implement the interfaces described to extend the functionality.

Cache

This is a list of the cache classes and interfaces, along with a brief description. See the online .NET API documentation for more information.

- **`CacheFactory`**—This class creates a `Cache` instance based on an instance of `DistributedSystem`. If a `cache.xml` file is specified by `DistributedSystem`, the cache is initialized based on the declarations loaded from that file. If a `cache.xml` file is used to create a cache and some of the regions already exist, then a warning states that the regions exist and the cache is created.
- **`Cache`**—This class is the entry point to the GemFire caching API. This class allows you to create regions. The cache is created by calling the `create` function of the `CacheFactory` class. When creating a cache, you specify a `DistributedSystem` that tells the new cache where to find other caches on the network and how to communicate with them.
- **`IGFSerializable`**—This interface is the superclass of all user objects in the cache that can be serialized and stored in the cache. GemFire provides built-in serializable types. See Table 4.2 on page 93 for a list of built-in types. Any custom types defined for objects that need to be transmitted as values should implement this interface. See Example 5.9 on page 125 for a code example.

- **`ICacheableKey`**—This interface derives from `IGFSerializable` and must be implemented by a type used as a key. It contains a `HashCode` and an `Equals(ICacheableKey)` function that place the value in an appropriate slot in the internal hash tables. The built-in types `CacheableString` and `CacheableInt32` implement this interface.
- **`Serializable`**—This class wraps the native C++ `gemfire::Serializable` objects as managed `IGFSerializable` objects. Whenever native C++ clients and .NET clients interoperate and are part of the same distributed system, the user-defined types that are `put` by the native C++ clients that have not been defined in .NET are returned as objects of this class.

  The API contains overloads for most of the `Region` methods and other methods which take `Serializable` as a value and are more optimized than the more generic `IGFSerializable` overloads. The application prefers using these overloads whenever the base class of an object is `Serializable` (for built-in types `CacheableString`, `CacheableBytes`, and `CacheableInt32`).
- **`CacheableKey`**—This class wraps the native C++ `gemfire::CacheableKey` objects as managed `ICacheableKey` objects for interoperability between C++ and .NET clients. Like the `Serializable` class, the application prefers using the overloads that take `CacheableKey` as arguments when using the built-in types `CacheableString` and `CacheableInt32`.
- **`DataInput`**—Supplies operations for reading primitive data values and user-defined objects from a byte stream.
- **`DataOutput`**—Provides operations for writing primitive data values and user-defined objects implementing `IGFSerializable` to an integer.
- **`Log`**—Defines methods available to clients that want to write a log message to their GemFire system shared log file. Any attempt to use an instance after its connection is disconnected throws a `NotConnectedException`. For any logged message the log file will contain:
  - The log level of the message.
  - The time the message was logged.
  - The ID of the connection and thread that logged the message.
  - The message itself, possibly with an exception including its stack trace.

Log Message Levels

A message always has a level, and logging levels are ordered. Enabling logging at a given level also enables logging at higher levels. The higher the level, the more important and urgent the message. The levels, in descending order of severity, are:

- **Error** (highest severity) is a message level indicating a serious failure. In general, error messages describe events that are of considerable importance and will prevent normal program execution.
- **Warning** is a message level indicating a potential problem. In general, warning messages describe events that are of interest to end users or system managers, or that indicate potential problems.
- **Info** is a message level for informational messages. Typically, these types of messages should be reasonably significant and should make sense to end users and system administrators.
- **Config** is a message level for static configuration messages, and are intended to provide a variety of static configuration information to assist in debugging problems that may be associated with particular configurations.
- **Fine** is a message level providing tracing information. The fine level should be used for information that will be broadly interesting to developers. This level is for the lowest volume and most important tracing messages.
- **Finer** indicates a moderately detailed tracing message.
- **Finest** indicates a very detailed tracing message. Logging calls for entering, returning, or throwing an exception are traced at the finest level.
- **Debug** (lowest severity) indicates a highly detailed tracing message. The debug level should be used for the most voluminous tracing messages.

Region

This is a list and brief description of the region classes and interfaces. See the online .NET API documentation for more information.

- **`Region`**—This class provides functions for managing regions and cached data. The functions for this class allow you to perform the following actions:
  - Retrieve information about the region, such as its parent region and region attribute objects.
  - Invalidate or destroy the region.
  - Create, update, invalidate and destroy region entries.
  - Determine, individually or as entire sets, the region's entry keys, entry values and `RegionEntry` objects.
- **`RegionEntry`**—This class contains the key and value for the entry, and provides all non-distributed entry operations. The operations of this object are not distributed and do not affect statistics.

Region Attributes

- **`RegionAttributes`**—This class holds all attribute values for a region and provides functions for retrieving all attribute settings. This class can only be modified by the `AttributesFactory` class before region creation, and the `AttributesMutator` class after region creation.
- **`AttributesMutator`**—This class allows modification of an existing region's attributes for application plug-ins and expiration actions. Each region has an `AttributesMutator` instance.
- **`Properties`**—Provides a collection of properties, each of which is a key/value pair. Each key is a string, and the value can be a string or an integer.

Application Callback Interfaces

- **`ICacheLoader`**—Application interface for loading data into a region.
- **`ICacheWriter`**—Application interface for synchronously handling region and entry events before the events occur. Entry events are `create`, `update`, `invalidate`, and `destroy`. Region events are `invalidate` and `destroy`. This class has the ability to abort events.
- **`ICacheListener`**—Listeners receive notifications when entries in a region change, or when changes occur to the region attributes themselves. The `ICacheListener` is a plug-in class for handling region and entry events after they occur. Entry events are `create`, `update`, `invalidate`, and `destroy`. Region events are `invalidate` and `destroy`.

The methods on `ICacheListener` are invoked asynchronously. Multiple events can cause concurrent invocation of `ICacheListener` methods. If event A occurs before event B, there is no guarantee that their corresponding `ICacheListener` method invocations will occur in the same order. Any exceptions thrown by the listener are caught by GemFire and logged.

It is important to ensure that minimal work is done in the listener before returning control back to GemFire. For example, a listener implementation may choose to hand off the event to a thread pool that processes the event on its thread rather than the listener thread

Events

- **`RegionEvent`**—This class provides information about the event, such as what region the event originated in, whether the event originated in a cache remote to the event handler, and whether the event resulted from a distributed operation.
- **`EntryEvent`**—This class provides all of the information available for the `RegionEvent`. It also provides entry-specific information, such as the old and new entry values and whether the event resulted from a `load` operation.

5.3 C++ Class to .NET Class Mappings

Wherever the native C++ class methods use pass-by-reference semantics to return data, the corresponding .NET methods shown in the following table return the object instead of using pass-by-reference semantics.

Table 5.1 C++ Class to .NET Class Mappings

C++ Class	.NET Class
class `gemfire::AttributesFactory`	sealed class `AttributesFactory`
class `gemfire::AttributesMutator`	sealed class `AttributesMutator`
class `gemfire::Cache`	sealed class `Cache`
abstract class `gemfire::Cacheable`	interface `IGFSerializable`
class `gemfire::CacheableBytes`	sealed class `CacheableBytes`; returns a copy of the internal buffer in `CacheableBytes.`Value unlike the native one that returns a pointer to the internal buffer.
class `gemfire::Cacheableint32`	sealed class `CacheableInt32`
class `gemfire::CacheableString`	sealed class `CacheableString`; returns a copy of the internal data in `CacheableString.`Value unlike the native one that returns a pointer to the internal string.
abstract class `gemfire::CacheableKey`	interface `ICacheableKey` plus wrapper `CacheableKey` class for native `CacheableKey` objects.
abstract class `gemfire::CacheListener`	interface `ICacheListener`
class `gemfire::CacheLoader`	interface `ICacheLoader` plus static class `CacheLoader`
class `gemfire::CacheWriter`	interface class `ICacheWriter`
class `gemfire::CacheFactory`	sealed class `CacheFactory`
class `gemfire::DataInput`	sealed class `DataInput`; the `DataInput` methods return the value instead of pass by reference of the native class, so the native overloaded `readInt` methods are mapped to `ReadInt16`, `ReadInt32`, and `ReadInt64`. Does not implement `currentBufferPosition` of the native class that returned a pointer to the internal buffer.
class `gemfire::DataOutput`	sealed class `DataOutput`; for clarity and consistency with `DataInput`, the native overloaded `writeInt` methods are mapped to `WriteInt16`, `WriteInt32`, and `WriteInt64`; the `GetBuffer` function returns a copy of the internal buffer, unlike the native one that returns a pointer to the internal buffer.
class `gemfire::DiskPolicyType`	enum `DiskPolicyType` plus static class `DiskPolicy` containing convenience methods for `DiskPolicyType` enumeration
class `gemfire::DistributedSystem`	sealed class `DistributedSystem`
class `gemfire::EntryEvent`	sealed class `EntryEvent`
class `gemfire::Exception`	class `GemfireException`
all other exceptions deriving from `gemfire::Exception`	corresponding exceptions deriving from `GemfireException`

C++ Class	.NET Class
class `gemfire::ExpirationAction`	enum `ExpirationAction` plus static class `Expiration` containing convenience methods for `ExpirationAction` enumeration
class `gemfire::Log`	static class `Log`; the native `Log::log` method is mapped to `Log.Write` to avoid the conflict with the class name which is reserved for the constructors of `Log` class; the various `loglevel Throw` or `Catch` methods are not implemented, since they are redundant to `Log::Log`, `Log::LogThrow`, and `Log::LogCatch` methods that take `LogLevel` as a parameter.
enum `gemfire::MemberType`	enum `MemberType`
abstract class `gemfire::PersistanceManager`	not provided; the user can register a C++ implementation using `AttributesFactory.SetPersistenceManager` but cannot implement a new one in .NET
class `gemfire::Properties`	sealed class `Properties`
class `gemfire::Properties::Visitor`	delegate `PropertiesVisitor`
abstract class `gemfire::Region`	sealed class `Region`. A new `Region` implementation cannot be created in .NET
class `gemfire::RegionAttributes`	sealed class `RegionAttributes`
class `gemfire::ScopeType`	enum `ScopeType` plus static class `Scope` containing convenience methods for `ScopeType` enumeration+
abstract class `gemfire::Serializable`	interface `IGFSerializable` plus wrapper `Serializable` class for native `Serializable`, `Cacheable`, and `UserData` objects. The native `toString` method is not provided, since the `ToString` method of the base object class provides the same functionality.
class `gemfire::SystemProperties`	sealed class `SystemProperties`
class `gemfire::UserData`	interface `IGFSerializable`
class `gemfire::VectorT<T>`	array of the given type, such as `T[]`

5.4 Object Lifetimes

The .NET API provides a managed set of assemblies for the C++ API. The underlying C++ API employs reference counting using smart pointers for most classes. This means that all API operations with those objects return a reference to the underlying object and not a copy. Consequently, the underlying object will not be freed as long as the .NET application holds a reference to an object. In other words, the underlying object will stay in memory until the .NET object is garbage-collected. As long as a reference to an object is alive, the artifacts it maintains will also be alive. For example, as long as a `Region` object is not garbage-collected, then the destructor of the C++ native persistence manager (if any) for the region is not invoked.

In the C++ API, the references to an object are reduced when the object goes out of scope for stack allocation, or is deleted explicitly for heap allocation. The object is destroyed when its reference count reaches zero. In the .NET API, the references are reduced when the object is garbage-collected or is explicitly disposed with the .NET `using` statement.

Since a reference to the objects is returned, any change to the object also immediately changes the object as stored internally. For instance, if an object of the user-defined class `BankAccount` is put into the cache using `Region.Put`, then a reference of that object is stored in the internal structures. If you modify the object, then the internal object also changes. However, it is not distributed to other members of the distributed system until another `Region.Put` is performed.

Objects of these classes use reference counting (the underlying C++ object is reference-counted):

- `AttributesMutator`
- `CacheableBytes`
- `CacheableInt32`
- `CacheableString`
- `Cache`
- `CacheStatistics`
- `DistributedSystem`
- `MemberId`
- `Properties`
- `RegionAttributes`
- `RegionEntry`
- `Region`
- `Serializable`

The following classes do not use reference counting:

- `AttributesFactory`
- `DataInput`
- `DataOutput`
- `EntryEvent`
- `RegionEvent`
- `SystemProperties`

All classes implementing an interface (`IGFSerializable`, `ICacheableKey`, `ICacheLoader`, `ICacheListener`, `ICacheWriter`) are reference-counted.

5.5 AppDomains

`AppDomains` are the units of isolation, security boundaries, and loading and unloading for applications in the .NET runtime. Multiple application domains can run in a single process. Each can have one or many threads, and a thread can switch application domains at runtime. The .NET managed assemblies make the assumption that any interface methods invoked by the native C++ layer are in the same `AppDomain` as that of the .NET DLL, otherwise an exception is thrown because it is unable to cross `AppDomain` boundaries.

Problem Scenarios

These scenarios describe processes and implementations that should be avoided when using `AppDomains`.

Using Application Callbacks

In this scenario, a .NET thread loads the GemFire DLL in application domain `AD1`. This thread may have access to the other domains in the application if code access security allows it. This thread can then call `AppDomain.CreateInstance` to create a callback object (`ICacheListener`, `ICacheLoader`, or `ICacheWriter`) in another domain called `AD2`. If the callback object is marshalled by reference, then the callback is executed in the domain where it is created (`AD2`). The thread that loads the GemFire DLL in domain `AD1` runs the callback methods in the second domain, `AD2`. An exception is thrown when the callback method is invoked because the native code that invokes the callback is not allowed to cross the `AppDomain` boundary.

Resolution: When an application creates and unloads application domains it should ensure that the application domain where the GemFire .NET DLL is loaded is the same domain where the application callback and `IGFSerializable` objects are created.

Loading an Application DLL in Multiple AppDomains

In this scenario, the application loads the GemFire DLL in one application domain, then reloads the GemFire DLL in another application domain (with or without unloading the previous `AppDomain`). The callbacks, as well as other interface implementations (`ICacheableKey`, `IGFSerializable`), throw exceptions because the native C++ code does not know about `AppDomains` and is loaded only once in the initial `AppDomain`.

Resolution: The application should always use the first `AppDomain` to load the GemFire DLL, or it should not load the GemFire DLL multiple times.

Native Client inside IIS

When you deploy more than one web application inside an Internet Information Service (IIS), the IIS creates an appdomain subprocess for each web application in the single process, but the native client C++ cache instance remains a singleton in the process. Because of this, you can run into conflicts between cache creation and closure by the different appdomains. For example, if one appdomain calls `cache.close`, it closes the cache for the entire process. Any further cache access operations by the other appdomains return cache closed exceptions. To manage this, Cache create/close provides reference counting of Cache create and close. Each process can use the counter to make sure it creates the Cache once and closes it once. To enable this, set the GemFire system property, `appdomain-enabled` to `true`.

5.6 Creating a Cache

The code snippets in this section provide examples of these basic programming operations:

- Creating a cache, which initializes the distributed system connection and local cache.
- Creating a cache using a `cache.xml` file.

In the following example, the operation connects to the distributed system and turns the application into a distributed system member.

Example 5.1 Connecting and Creating the Cache

```
CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(null);
   Cache cache = cacheFactory.Create();
```

For systems with security enabled, the credentials for a joining member are authenticated when the cache is created and the system connection is made. See *Security* on page 163 for more information about secure connections to a distributed system.

In the next example, the application creates the cache by calling the `CacheFactory.Create` function with the new `DistributedSystem` object.

A cache can also be created by referencing a `cache.xml` file, as shown in the following example.

Example 5.2 Creating a Cache with a cache.xml File

```
Properties prop = Properties.Create();
prop.Insert("cache-xml-file", "cache.xml");
CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(prop);
Cache cache = cacheFactory.Create();
```

See Chapter 3, *Cache Initialization File* for more information about the `cache.xml` file.

5.7 Creating a Region with Caching and LRU

Region data can be stored to disk using the LRU overflow process to satisfy region capacity restrictions without completely destroying the local cache data. The storage mechanism uses disk files to hold region entry data. When an entry is overflowed, its value is written to disk but its key and entry object remain in the cache.

In the next example, the application overrides the default region attributes where necessary.

- The `SetLruEntriesLimit` function sets the maximum number of entries to hold in a caching region to `20000`.
- The `SetInitialCapacity` function sets the number of entries a caching region should initially expect to hold to `20000`.

For all other region attributes, the defaults are used.

Example 5.3 Creating a Region with Caching and LRU

```
RegionFactory regionFact =
   cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
region = regionFact.SetLruEntriesLimit(20000)
   .SetInitialCapacity(20000)
   .Create("exampleRegion");
```

Cache Overflow

By default, the cache is kept in memory. When `LruEntriesLimit` is exceeded, those LRU entries are destroyed. With cache overflow, the over-limit LRU entry values are written to disk, keeping the key in memory. When one of those entries is requested, its value is loaded back into memory. The disk policy specifies whether to destroy or write overflow entries. See DiskPolicy (page 57) for more information.

A persistence manager API is provided with the GemFire Enterprise native client for customers who need to provide their own mechanism for writing to disk or to a backend database. The persistence manager uses the open source Berkeley DB library. A native C++ API library implementation of this API can be registered and used with the .NET API. The Berkeley DB API is provided in `bdbimpl.dll` in the native client `\bin` directory. See Appendix A, *Installing the Berkeley DB Persistence Manager*, on page 289 for information about installing and configuring Berkeley DB.

This example creates a region with disk overflow.

Example 5.4 Creating a Region With Disk Overflow

```
Properties bdbProperties = Properties.Create();
bdbProperties.Insert("PersistenceDirectory", "BDB");
bdbProperties.Insert("EnvironmentDirectory", "BDBEnv");
bdbProperties.Insert("CacheSizeGb", "1");
bdbProperties.Insert("CacheSizeMb", "10");
bdbProperties.Insert("PageSize", "65536");
bdbProperties.Insert("MaxFileSize", "512000");

RegionFactory regionFact =
   cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
region = regionFact.SetLruEntriesLimit(20000)
   .SetInitialCapacity(20000)
   .SetDiskPolicy(DiskPolicyType.Overflows)
   .SetPersistenceManager("BDBImpl", "createBDBInstance", bdbProperties)
   .Create("exampleRegion");
```

5.8 Adding an Entry to the Cache

A native client region can be populated with cache entries using the .NET `Region.Put` or the `Region.Create` API functions. The `Put` function places a new value into a region entry with the specified key, while the `Create` function creates a new entry in the region. Both functions provide a user-defined parameter object to any cache writer invoked in the process, and new values for both functions are propagated to a connected cache server.

When the `Put` function adds an entry, the previous value is overwritten if there is already an entry associated with the specified key in the region.

By default, string keys or values are implicitly converted to `CacheableString` when `Put` and `Create` are called for the `Region` API. Similarly, integer keys or values are implicitly converted to `CacheableInt32` and byte array values are converted to `CacheableBytes`.

In the next example, the program puts 100 entries into the cache by iteratively creating keys and values, both of which are integers.

Example 5.5 Using the API to Put Values Into the Cache

```
for (int i = 0; i < 100; i++) {
   region.Put(i, i);
}
```

5.9 Accessing an Entry

The standard .NET `Region.Get` API method returns the value associated with the specified key, and passes the callback argument to any cache loaders or cache writers that are invoked in the operation. If the value is not present locally then it is requested from the cache server. If the cache server request is unsuccessful then a local cache loader is invoked.

The entry value is either retrieved from the local cache or fetched by the region's cache loader.

In the following example, the program does a `Get` for each entry that was `Put` into the cache in the previous example.

Example 5.6 Using the Get API to Retrieve Values From the Cache

```
for (int i = 0; i < 100; i++) {
   CacheableInt32 value = region.Get(i) as CacheableInt32;
}
```

5.10 Serialization

The GemFire Enterprise native client .NET API provides an `IGFSerializable` interface for fast and compact data serialization. This section discusses serialization and presents a simple implementation example.

Serializable Types

The native client provides a set of built-in serializable types that are automatically registered at initialization. See Table 4.2 on page 93 for a list of the built-in types and their descriptions.

If your application uses more complex key types that you want to make more accessible or easier to handle, you can derive a new class from `IGFSerializable`. Another option is for the application to do its own object serialization using the GemFire `CacheableBytes` type or a custom type.

The GemFire `IGFSerializable` *interface does not support object graphs with multiple references to the same object. If your application uses these types of circular graphs, you must address this design concern explicitly.*

Serialization Classes

The native client .NET API also provides two classes for convenient wrapping of generic objects as `IGFSerializable` values: `CacheableObject` and `CacheableObjectXml`.

CacheableObject with BinaryFormatter

`CacheableObject` uses the .NET runtime serialization with `BinaryFormatter` to serialize and deserialize objects. The rules that apply to .NET runtime serialization also apply to the `CacheableObject` class. That class, along with the classes of all the contained objects, should either be marked with the `Serializable` attribute or implement the `System.Runtime.Serialization.ISerializable` interface. See the online .NET API documentation for more details.

CacheableObjectXml with XmlSerializer

`CacheableObjectXml` uses the .NET `XmlSerializer` class to serialize and deserialize objects. By default, the .NET `XmlSerializer` serializes the public members and accessors of the class. This behavior can be altered using attributes. Alternatively, the application can implement the `System.Xml.Serialization.IXmlSerializable` interface. Refer to the online .NET API documentation for details of the various serialization attributes and the `IXmlSerializable` interface.

The .NET runtime serialization and `XmlSerializer` *have reduced performance compared to a custom* `IGFSerializable` *implementation because they need to use reflection at runtime to serialize or deserialize. An application should always implement the* `IGFSerializable` *interface over either the* `ISerializable` *or* `IXmlSerializable` *interfaces.*

How Serialization Works

When your application puts an object into the cache for distribution, GemFire serializes the data by taking these steps:

1. Call the appropriate `ClassId` function and create the `TypeId` from it.
2. Write the `TypeId` for the instance
3. Invoke the `ToData` function for the instance.

When your application subsequently receives a byte array, GemFire take the following steps:

1. Decode the `TypeId` and create an object of the designated type, using the registered factory functions.
2. Invoke the `FromData` function with input from the data stream.
3. Decode the data and then populate the data fields.

The `TypeId` is an integer of four bytes, which is a combination of `ClassId` integer and `0x27`, which is an indicator of user-defined type.

Implementing the IGFSerializable Interface

In order to store your own data types in the cache, you must implement the GemFire `IGFSerializable` interface using these steps:

1. Implement the `ToData` function that serializes your data:

   ```
   void ToData(DataOutput output)
   ```

   The `ToData` function is responsible for copying all of the data fields for the object to the object stream. The `DataOutput` class represents the output stream and provides methods for writing the primitives in a network byte order. For more about this, see the online API documentation for `DataOutput`.

2. Implement the `FromData` function that consumes a data input stream and repopulates the data fields for the object:

   ```
   void fromData (DataInput& input)
   ```

   The `DataInput` class represents the input stream and provides methods for reading input elements. The `FromData` function must read the elements of the input stream in the same order that they were written by `ToData`. For more about this, see the online API documentation for `DataInput`.

3. Implement the `ClassId` function to return an integer which is unique for your class (in the set of all of your user-defined classes).

The next example shows a simple class, `BankAccount`, that encapsulates two `ints`: `customerId` and `accountId`.

Example 5.7 The Simple BankAccount Class

```
public class BankAccount
{
   private int m_customerId;
   private int m_accountId;
   public int Customer
   {
      get
      {
         return m_customerId;
```

```
            }
        }
        public int Account
        {
            get
            {
                return m_accountId;
            }
        }
        public BankAccount(int customer, int account)
        {
            m_customerId = customer;
            m_accountId = account;
        }
    }
```

To make `BankAccount` serializable, you implement the `IGFSerializable` interface as shown in the following example:

Example 5.8 Implementing a Serializable Class

```
public class BankAccount : IGFSerializable
    {
      private int m_customerId;
      private int m_accountId;
      public int Customer
      {
        get
        {
          return m_customerId;
        }
      }
      public int Account
      {
        get
        {
          return m_accountId;
        }
      }
      public BankAccount(int customer, int account)
      {
        m_customerId = customer;
        m_accountId = account;
      }
      // Our TypeFactoryMethod
      public static IGFSerializable CreateInstance()
      {
        return new BankAccount(0, 0);
      }
      #region IGFSerializable Members
      public void ToData(DataOutput output)
      {
        output.WriteInt32(m_customerId);
        output.WriteInt32(m_accountId);
      }
      public IGFSerializable FromData(DataInput input)
      {
        m_customerId = input.ReadInt32();
        m_accountId = input.ReadInt32();
        return this;
      }
      public UInt32 ClassId
      {
        get
        {
          return 11;
        }
      }
      public UInt32 ObjectSize
      {
        get
        {
          return (UInt32)(sizeof(Int32) + sizeof(Int32));
        }

      }
    }
```

Registering the Type

To be able to use the `BankAccount` type, you must register it with the type system so that when an incoming stream contains a `BankAccount`, it can be manufactured from the associated `TypeFactoryMethod`.

```
Serializable.RegisterType(BankAccount.CreateInstance);
```

Typically, you would register the type before calling the function `DistributedSystem.Connect`.

Using ClassId

A `ClassId` is an integer that returns the `ClassId` of the instance being serialized. The `ClassId` is used by deserialization to determine what instance type to create and deserialize into.

Using DSFID

A `DSFID` is an integer that returns the data serialization fixed ID type. `DSFID` is used to determine what instance type to create and deserialize into. `DSFID` should not be overridden by custom implementations, and it is reserved only for built-in serializable types.

Using Custom Key Types

If your application uses key types that are too complex to easily force into `CacheableString`, you can likely improve performance by deriving a new class from `ICacheableKey` and implementing `HashCode` and `Equals(ICacheableKey)` functions. For example, if you have hybrid data types such as floating point numbers, you can implement your own derivation of `ICacheableKey` that encapsulates the floating point number. Comparing floating point numbers in this way provides greater performance than comparing a string representation of the floating point numbers, with such noticeable improvements as faster cache access and smaller payloads.

See *Serialization in Native Client Mode with a Java Server* on page 99 for information about implementing key types for a native client that is used with a Java cache server.

To extend a type that implements `IGFSerializable` to be a key, add the extra `HashCode` and `Equals(ICacheableKey)` methods in `ICacheableKey`.

The following example shows how to extend an `IGFSerializable` class to be a key. Example 5.10 on page 128 shows how to use the `BankAccount` custom key type.

Example 5.9 Extending an IGFSerializable Class to Be a Key

```
class BankAccountKey : ICacheableKey
    {
      #region Private members

      private int m_customerId;
      private int m_accountId;

      #endregion

      #region Public accessors

      public int Customer
      {
        get
        {
          return m_customerId;
        }
      }

      public int Account
      {
        get
        {
          return m_accountId;
        }
      }

      #endregion

      public BankAccountKey(int customer, int account)
      {
        m_customerId = customer;
        m_accountId = account;
      }

      // Our TypeFactoryMethod
      public static IGFSerializable CreateInstance()
      {
        return new BankAccountKey(0, 0);
      }

      #region IGFSerializable Members

      public void ToData(DataOutput output)
      {
        output.WriteInt32(m_customerId);
        output.WriteInt32(m_accountId);
      }

      public IGFSerializable FromData(DataInput input)
      {
        m_customerId = input.ReadInt32();
        m_accountId = input.ReadInt32();
        return this;
      }

      public UInt32 ClassId
      {
        get
        {
          return 11;
        }
      }

      public UInt32 ObjectSize
      {
        get
        {
          return (UInt32)(sizeof(Int32) + sizeof(Int32));
        }

      }

      #endregion

      #region ICacheableKey Members

      public bool Equals(ICacheableKey other)
      {
        BankAccountKey otherAccount = other as BankAccountKey;
        if (otherAccount != null) {
          return (m_customerId == otherAccount.m_customerId) &&
            (m_accountId == otherAccount.m_accountId);
        }
        return false;
      }

      public override int GetHashCode()
      {
        return (m_customerId ^ m_accountId);
      }

      #endregion

      #region Overriden System.Object methods

      public override bool Equals(object obj)
      {
        BankAccountKey otherAccount = obj as BankAccountKey;
        if (otherAccount != null) {
          return (m_customerId == otherAccount.m_customerId) &&
            (m_accountId == otherAccount.m_accountId);
        }
        return false;
      }

      // Also override ToString to get a nice string representation.
      public override string ToString()
      {
        return string.Format("BankAccountKey( customer: {0}, account: {1}
)",
          m_customerId, m_accountId);
      }

      #endregion
    }
```

Handling Object Graphs

If you have a graph of objects in which each node can be serializable, the parent node calls `DataOutput.WriteObject` to delegate the serialization responsibility to its child nodes. Similarly, your application calls `DataInput.ReadObject` to deserialize the object graph.

For more information, see the online API documentation for `DataOutput` and `DataInput`.

Handling Data as a Blob

If you have data that is best handled as a blob, such as structs that do not contain pointers, use the serializable type `CacheableBytes`. `CacheableBytes` is a blob class that implements the serialization for you.

`CacheableBytes` also provides direct access to the blob data. It cannot be used as a key, but it enables you to modify data in place and then put it into the region again to distribute the change.

For more information, see the online API documentation for `CacheableBytes`.

5.11 Using a Custom Class

The next example shows how to use the `BankAccount` custom key type and the `AccountHistory` value type that were previously defined.

Example 5.10 Using a BankAccount Object

```
class AccountHistory : IGFSerializable
    {
      #region Private members

      private List<string> m_history;

      #endregion

      public AccountHistory()
      {
        m_history = new List<string>();
      }

      public void ShowAccountHistory()
      {
        Console.WriteLine("AccountHistory:");
        foreach (string hist in m_history) {
          Console.WriteLine("\t{0}", hist);
        }
      }

      public void AddLog(string entry)
      {
        m_history.Add(entry);
      }

      public static IGFSerializable CreateInstance()
      {
        return new AccountHistory();
      }

      #region IGFSerializable Members

      public IGFSerializable FromData(DataInput input)
      {
        int len = input.ReadInt32();

        m_history.Clear();
        for (int i = 0; i < len; i++) {
          m_history.Add(input.ReadUTF());
        }
        return this;
      }

      public void ToData(DataOutput output)
      {
        output.WriteInt32(m_history.Count);
        foreach (string hist in m_history) {
          output.WriteUTF(hist);
        }
      }
```

```
          public UInt32 ClassId
          {
            get
            {
              return 0x05;
            }
          }

          public UInt32 ObjectSize
          {
            get
            {
              UInt32 objectSize = 0;
              foreach (string hist in m_history) {
                objectSize += (UInt32)(hist == null ? 0 : sizeof(char) * hist.Length);
              }
              return objectSize;

            }

          }

          #endregion
        }
          public class TestBankAccount
          {
            public static void Main()
            {
              // Register the user-defined serializable type.
              Serializable.RegisterType(AccountHistory.CreateInstance);
              Serializable.RegisterType(BankAccountKey.CreateInstance);
              // Create a cache.
              CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(null);
              Cache cache = cacheFactory.Create();
              // Create a region.
              RegionFactory regionFactory =
cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
              Region region = regionFactory.Create("BankAccounts");
              // Place some instances of BankAccount cache region.
              BankAccountKey baKey = new BankAccountKey(2309, 123091);
              AccountHistory ahVal = new AccountHistory();
              ahVal.AddLog("Created account");
              region.Put(baKey, ahVal);
              Console.WriteLine("Put an AccountHistory in cache keyed with
BankAccount.");
              // Display the BankAccount information.
              Console.WriteLine(baKey.ToString());
              // Call custom behavior on instance of AccountHistory.
              ahVal.ShowAccountHistory();
              // Get a value out of the region.
              AccountHistory history = region.Get(baKey) as AccountHistory;
              if (history != null)
              {
                Console.WriteLine("Found AccountHistory in the cache.");
                history.ShowAccountHistory();
                history.AddLog("debit $1,000,000.");
                region.Put(baKey, history);
                Console.WriteLine("Updated AccountHistory in the cache.");
              }
              // Look up the history again.
              history = region.Get(baKey) as AccountHistory;
              if (history != null)
              {
                Console.WriteLine("Found AccountHistory in the cache.");
                history.ShowAccountHistory();
              }
              // Close the cache.
              cache.Close();
            }
          }

        //Example 5.12 Using ICacheLoader to Load New Integers in the Region
        class ExampleLoaderCallback : ICacheLoader
        {
          #region Private members
          private int m_loads = 0;
          #endregion
          #region Public accessors
          public int Loads
          {
            get
            {
              return m_loads;
            }
          }
          #endregion
          #region ICacheLoader Members
          public IGFSerializable Load(Region region, ICacheableKey key,
    IGFSerializable helper)
          {
            return new CacheableInt32(m_loads++);
          }
          public virtual void Close(Region region)
          {
            Console.WriteLine("Received region close event.");
          }
          #endregion
        }
```

5.12 Application Callbacks

For region-level events, an application can use `AttributesFactory.SetCache*` methods to implement and register the `ICacheLoader`, `ICacheWriter`, and `ICacheListener` interfaces to perform custom actions.

You can use `Region.Put` for simple caching situations. For more complex needs, you should implement the `ICacheLoader` interface and allow the cache to manage the creation and loading of objects. When a `Region.Get` is called for a region entry with a value of `null`, the `ICacheLoader::Load` method of the cache loader (if any) for the region is invoked. A static `CacheLoader::NetSearch` method is provided which can be used by `ICacheLoader` implementations to locate the requested key in the distributed system. The `ICacheListener` interface can be used to listen to various region events after events such as `create`, `update`, or `invalidate` of region entries have occurred. The `ICacheWriter` interface is invoked before the events have occurred. The following example demonstrates an `ICacheLoader` implementation for loading new integers into a region.

Example 5.11 Using ICacheLoader to Load New Integers in the Region

```
class ExampleLoaderCallback : ICacheLoader
    {
      #region Private members
      private int m_loads = 0;
      #endregion
      #region Public accessors
      public int Loads
      {
        get
        {
          return m_loads;
        }
      }
      #endregion
      #region ICacheLoader Members
      public IGFSerializable Load(Region region, ICacheableKey key,
IGFSerializable helper)
      {
        return new CacheableInt32(m_loads++);
      }
      public virtual void Close(Region region)
      {
        Console.WriteLine("Received region close event.");
      }
      #endregion
    }
```

The next example shows how to implement `ICacheWriter` to track `create` and `update` events for a region.

Example 5.12 Using ICacheWriter to Track Creates and Updates for a Region

```
class ExampleWriterCallback : ICacheWriter
    {
      #region Private members
      private int m_creates = 0;
      private int m_updates = 0;
      #endregion
      #region Public accessors
      public int Creates
      {
        get
        {
          return m_creates;
        }
      }
      public int Updates
      {
        get
        {
          return m_updates;
        }
      }
      #endregion
      public void ShowTallies()
      {
        Console.WriteLine("Updates = {0}, Creates = {1}",
        m_updates, m_creates);
      }
      #region ICacheWriter Members
      public virtual bool BeforeCreate(EntryEvent ev)
      {
        m_creates++;
        Console.WriteLine("Received BeforeCreate event of: {0}", ev.Key);
        return true;
      }
      public virtual bool BeforeDestroy(EntryEvent ev)
      {
        Console.WriteLine("Received BeforeDestroy event of: {0}", ev.Key);
        return true;
      }
      public virtual bool BeforeRegionDestroy(RegionEvent ev)
      {
        Console.WriteLine("Received BeforeRegionDestroy event of: {0}", ev.Region.Name);
        return true;
      }
      public virtual bool BeforeUpdate(EntryEvent ev)
      {
        m_updates++;
        Console.WriteLine("Received BeforeUpdate event of: {0}", ev.Key);
        return true;
      }
      public virtual bool BeforeRegionClear(RegionEvent ev)
      {
        Console.WriteLine("Received BeforeRegionClear event");
        return true;
      }
      public virtual void Close(Region region)
      {
        Console.WriteLine("Received Close event of: {0}", region.Name);
      }
      #endregion
    }
```

This example demonstrates an implementation of `ICacheListener`.

Example 5.13 A Sample ICacheListener Implementation

```
class ExampleListenerCallback : ICacheListener
    {
      #region ICacheListener Members
      public void AfterCreate(EntryEvent ev)
      {
        Console.WriteLine("Received AfterCreate event of: {0}", ev.Key);
      }
      public void AfterDestroy(EntryEvent ev)
      {
        Console.WriteLine("Received AfterDestroy event of: {0}", ev.Key);
      }
      public void AfterInvalidate(EntryEvent ev)
      {
        Console.WriteLine("Received AfterInvalidate event of: {0}",
        ev.Key);
      }
      public void AfterRegionDestroy(RegionEvent ev)
      {
        Console.WriteLine("Received AfterRegionDestroy event of region:
{0}",
        ev.Region.Name);
      }
      public void AfterRegionClear(RegionEvent ev)
      {
        Console.WriteLine("Received AfterRegionClear event of region: {0}",
        ev.Region.Name);
      }
      public void AfterRegionLive(RegionEvent ev)
      {
        Console.WriteLine("Received AfterRegionLive event of region: {0}",
        ev.Region.Name);
      }
      public void AfterRegionInvalidate(RegionEvent ev)
      {
        Console.WriteLine("Received AfterRegionInvalidate event of
region:{0}", ev.Region.Name);
      }
      public void AfterRegionDisconnected(Region region)
      {
        Console.WriteLine("Received AfterRegionDisconnected event of
region:{0}", region.Name);
      }
      public void AfterUpdate(EntryEvent ev)
      {
        Console.WriteLine("Received AfterUpdate event of: {0}", ev.Key);
      }
      public void Close(Region region)
      {
        Console.WriteLine("Received Close event of region: {0}",
        region.Name);
      }
      #endregion
    }
```

5.13 Examples

A Simple C# Example

This example shows how to connect to GemFire, create a cache and region, put and get keys and values, and disconnect.

Example 5.14 Simple C# Code

```
class FirstSteps
    {
      public static void Main()
      {
        // 1. Create a cache
        CacheFactory cacheFactory = CacheFactory.CreateCacheFactory();
        Cache cache = cacheFactory.Create();
        // 2. Create default region attributes using region factory
        RegionFactory regionFactory =
cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
        // 3. Create region
        Region region = regionFactory.Create("exampleputgetregion");
        // 4. Put some entries
        int iKey = 777;
        string sKey = "abc";
        region.Put(iKey, 12345678);
        region.Put(sKey, "testvalue");
        // 5. Get the entries
        CacheableInt32 ciValue = region.Get(iKey) as CacheableInt32;
        Console.WriteLine("Get - key: {0}, value: {1}", iKey,
ciValue.Value);
        CacheableString csValue = region.Get(sKey) as CacheableString;
        Console.WriteLine("Get - key: {0}, value: {1}", sKey,
csValue.Value);
        // 6. Close cache
        cache.Close();
      }
    }
```

5.14 Troubleshooting .NET Applications

The .NET Framework does not find managed DLLs using the conventional `PATH` environment variable. In order for your assembly to find and load a managed DLL, it must either be loaded as a private assembly using `assemblyBinding`, or it must be installed into the Global Assembly Cache (GAC). The GAC utility must be run on every machine that runs the .NET code.

If an assembly attempts to load the `GemStone.GemFire.Cache.dll` without meeting this requirement, you receive this `System.IO.FileNotFoundException`:

```
{{

Unhandled Exception: System.IO.FileNotFoundException: Could not load file
or assembly 'GemStone.GemFire.Cache, Version=1.2.0.0, Culture=neutral,
PublicKeyToken= 126e6338d9f55e0c' or one of its dependencies. The system
cannot find the file specified.
File name: 'GemStone.GemFire.Cache, Version=1.2.0.0, Culture=neutral,
PublicKeyT oken=126e6338d9f55e0c'
at HierarchicalClient.Main()

}}
```

Resolving the Error

Each computer where the common language runtime is installed has a machine-wide code cache called the Global Assembly Cache (GAC). The global assembly cache stores assemblies specifically designated to be shared by several applications on the computer. You should share assemblies by installing them into the global assembly cache only when you need to. As a general guideline, keep assembly dependencies private, and locate assemblies in the application directory unless sharing an assembly is explicitly required.

Using GemStone.GemFire.Cache.dll As a Private Assembly

To access `GemStone.GemFire.Cache.dll` as a private assembly, you need to specify a `.config` file for your application. The file needs to be the same name as your application, with a `.config` suffix. For example, the `.config` file for `main.exe` would be `main.exe.config`. The two files must reside in the same directory.

Follow these steps to create a `.config` file:

1. Copy `%GFCPP%/docs/default.exe.config` to the appropriate location.
2. Rename `default.exe.config` to the name of your application.
3. Change the `href` attribute of the `CodeBase` element to point to your `GemStone.GemFire.Cache.dll` file. Either http, relative, or absolute paths will work.

The following example shows an excerpt of a `.config` file. The `PublicKeyToken` value is only an example, and the `codebase version` value is not set correctly. See `%GFCPP%/docs/default.exe.config` for an actual example for this release.

Example 5.15 A Sample .config File

```
<configuration>
  <runtime>
   <assemblyBinding
      xmlns="urn:schemas-microsoft-com:asm.v1">
     <dependentAssembly>
       <assemblyIdentity name="GemStone.GemFire.Cache"
          publicKeyToken="126e6338d9f55e0c"
          culture="neutral" />
       <codeBase version="0.0.0.0"
          href="../../bin/GemStone.GemFire.Cache.dll"/>
     </dependentAssembly>
   </assemblyBinding>
 </runtime>
</configuration>
```

If the `.config` file contain errors, no warning or error messages are issued. The application runs as if no `.config` file is present.

Implementing the Shared Assembly

Follow these steps to install the shared assembly into the Global Assembly Cache (GAC):

1. Go to the `NativeClient_xxxx` directory.

   ```
   cd %GFCPP%
   ```

2. Run the GAC utility to install `GemStone.GemFire.Cache.dll` into the GAC.

   ```
   gacutil.exe /if GemStone.GemFire.Cache.dll
   ```

When you are ready to uninstall, use the `/u` switch. More information on the GAC utility can be found at http://www.msdn.com, or by using `gacutil.exe /?`.

Chapter 6

Setting Properties

This chapter describes the GemFire Enterprise native client configuration attributes that you can modify through the `gfcpp.properties` configuration file, and also shows a programmatic approach.

In this chapter:

6.1 Configuration Overview

A GemFire Enterprise native client or GemFire cache server needs to be configured to participate in the distributed system. Configuration settings can be made through the use of configuration files. An application or cache server process reads its configuration files at startup. In addition, applications can programmatically configure an instance that is connecting to the distributed system.

Each application and cache server also requires a GemFire license file. See Licensing on page 30 and the GemFire Enterprise System Administrator's Guide *for details on setting up licensing.*

Native Client Configuration

These two configuration files are available for configuring the native client:

- `gfcpp.properties` for native client system-level configuration. Most of this chapter explains how to configure the `gfcpp.properties` file.
- `cache.xml` for cache-level configuration. The configuration of the caches is part of the application development process and is covered in *Cache Initialization File* on page 75.

The cache-level configuration file is referred to as `cache.xml` *in this book, but the file name is configurable. Refer to the specific application documentation for the name of their XML file.*

About gfcpp.properties Configuration Files

The `gfcpp.properties` file provides any local settings required to connect a client to a distributed system, along with settings for licensing, logging, and statistics. Table 6.2 on page 145 describes the attributes and their default settings.

The application software may include a set of `gfcpp.properties` files. You set any attributes needed for the application design in these files, then you can add any attributes needed for the local site.

If you do not have `gfcpp.properties` files, use any text editor to create them. See Appendix B, *gfcpp.properties Example File*, on page 293 for a sample of the file format and contents.

Configuration File Location

A native client looks for `gfcpp.properties` first in the working directory where the process runs, then in *`productDir`*`/defaultSystem`. Use the `defaultSystem` directory to group configuration files or to share them among processes for more convenient administration. If `gfcpp.properties` is not found, the process starts up with the default settings.

For the `cache.xml` cache configuration file, a native client looks for the path specified by the `cache-xml-file` attribute in `gfcpp.properties` (see page 143). If `cache.xml` is not found, the process starts with an unconfigured cache.

Configuration Options

The typical configuration procedure for a native client includes the high-level steps listed below. The rest of this chapter provides the details.

1. Place the `gfcpp.properties` file for the application in the working directory or in *`productDir`*`/defaultSystem`. Use the configuration file that came with the application software if there is one, or create your own. See Appendix B, *gfcpp.properties Example File*, on page 293 for a sample of the file format and contents.

2. Place the `cache.xml` file for the application in the desired location and specify its path in the `gfcpp.properties` file.

3. Add other attributes to the `gfcpp.properties` file as needed for the local system architecture. See Table 6.2 on page 145 for the configurable attributes, and Appendix B, *gfcpp.properties Example File*, on page 293 for a sample of the file format.

Running a Native Client Out of the Box

If you start a native client without any configurations, it uses any attributes set programmatically plus any hard-coded defaults (listed in Table 6.2 on page 145). Running with the defaults is a convenient way to learn the operation of the distributed system and to test which attributes need to be reconfigured for your environment.

Cache Server Configuration

When a cache server joins a distributed GemFire system it must have certain configuration information set for proper operation. This information is defined in the GemFire property settings. The cache server specifies a set of configuration attributes at startup. These attributes can be specified in a number of ways, such as from the command line or in system properties, but they are generally provided in the server's `gemfire.properties` file.

Configuration File Location

For the GemFire cache server, the `gemfire.properties` file is usually stored in the current working directory. For more information, see the *GemFire Enterprise System Administrator's Guide*.

Modifying Attributes Outside the gemfire.properties File

In addition to the `gemfire.properties` file, you can pass attributes to the cache server on the command line. These override any settings found in the `gemfire.properties` file for starting the cache server.

For example, the attributes are passed in as `attName=attValue` pairs, such as `log-file=newLogFileName`. For more information, see the *GemFire Enterprise System Administrator's Guide*.

Attribute Definition Priority

Attributes can be specified in different ways, which leads to the possibility of conflicting definitions. Applications can be configured programmatically, and that has priority over other settings. Check your application documentation to see whether this applies in your case.

In case an attribute is defined in more than one place, the following table shows which configuration source is used.

Table 6.1 Priority of Configuration Settings for Native Clients and Cache Servers

Native Client	GemFire Cache Server
1. Programmatic configuration	1. `*workingDirectory*/gemfire.properties` file
2. GemFire properties set on the command line	2. `*productDir*/defaultSystem/gemfire.properties` file
3. `*workingDirectory*/gfcpp.properties` file	3. GemFire defaults
4. `*productDir*/defaultSystem/gfcpp.properties` file	
5. GemFire defaults	

The `gfcpp.properties` files and programmatic configuration are optional. If they are not present, no warnings or errors occur. For details on programmatic configuration through the `Properties` object, see *Defining Properties Programmatically* on page 149.

6.2 Attributes in gfcpp.properties

This section describes the `gfcpp.properties` settings that can be used when a native client connects to a distributed system. The property settings fall into these categories:

- **General Properties**—These properties provide basic information for the process, such as cache creation parameters.
- **Licensing Properties**—These properties supply product licensing information.
- **Logging Properties**—These properties provide instructions on how and where to log system messages.
- **Statistics Archiving Properties**—These properties provide instructions to collect and archive statistics information.
- **Durable Client Properties**—These properties provide information about the durable clients connected to the system.
- **Security Properties**—These properties provide information about various security parameters.

Alphabetical Lookup

- `archive-disk-space-limit` (page 147)
- `archive-file-size-limit` (page 147)
- `cache-xml-file` (page 145)
- `conflate-events` (page 145)
- `connection-pool-size` (page 145)
- `crash-dump-enabled` (page 145)
- `durable-client-id` (page 147)
- `durable-timeout` (page 147)
- `enable-time-statistics` (page 147)
- `grid-client` (page 145)
- `heap-lru-delta` (page 145)
- `heap-lru-limit` (page 145)
- `license-file` (page 146)
- `license-type` (page 146)
- `log-disk-space-limit` (page 146)
- `log-file` (page 146)
- `log-file-size-limit` (page 146)
- `log-level` (page 146)
- `max-socket-buffer-size` (page 145)
- `notify-ack-interval` (page 145)
- `notify-dupcheck-life` (page 145)
- `ping-interval` (page 146)
- `redundancy-monitor-interval` (page 146)
- `security-client-auth-factory` (page 148)
- `security-client-auth-library` (page 148)
- `security-client-dhalgo` (page 148)
- `security-client-kspath` (page 148)
- `stacktrace-enabled` (page 146)
- `statistic-sampling-enabled` (page 147)
- `statistic-archive-file` (page 147)
- `statistic-sampling-enabled` (page 147)

For the system properties that relate to high availability, see *Sending Periodic Acknowledgement* on page 153. For a list of security-related system properties and their descriptions, see Table 8.1 on page 174.

Attribute Definitions

This table lists GemFire configuration attributes that can be stored in the `gfcpp.properties` file to be read by a native client. The attributes are grouped according to the categories listed at the beginning of this section.

Table 6.2 Attributes in gfcpp.properties

`gfcpp.properties` Attribute	**Description**	**Default**
	General Properties	
`cache-xml-file`	The name and path of the file whose contents are used by default to initialize a cache if one is created. If not specified, the native client starts with an empty cache, which is populated at runtime. See Chapter 3, *Cache Initialization File*, on page 75 for more information on the cache initialization file.	no default
`heap-lru-delta`	When heap LRU is triggered, this is the amount that gets added to the percentage that is above the `heap-lru-limit` amount. LRU continues until the memory usage is below `heap-lru-limit` minus this percentage. This property is only used if `heap-lru-limit` is greater than `0`.	`10`
`heap-lru-limit`	Maximum amount of memory, in megabytes, used by the cache for all regions. If this limit is exceeded by `heap-lru-delta` percent, LRU reduces the memory footprint as necessary. If not specified, or set to `0`, memory usage is governed by each region's LRU entries limit, if any.	`0`
`conflate-events`	The client side conflation setting, which is sent to the server.	`server`
`connection-pool-size`	The number of connections per endpoint	`5`
`crash-dump-enabled`	Whether crash dump generation for unhandled fatal errors is enabled. True is enabled, false otherwise.	`true`
`grid-client`	If `true`, the client starts does not start various internal threads so that startup and shutdown time is reduced.	`false`
`max-socket-buffer-size`	The maximum size of the socket buffers, in bytes, that the native client will try to set for client-server connections.	`65 * 1024`
`notify-ack-interval`	The interval, in seconds, in which client sends acknowledgements for subscription notifications.	`1`
`notify-dupcheck-life`	The amount of time, in seconds, the client tracks subscription notifications before dropping the duplicates.	`300`

`gfcpp.properties` Attribute (Continued)	Description	Default
`ping-interval`	How often, in seconds, to communicate with the server to show the client is alive. Pings are only sent when the `ping-interval` elapses between normal client messages. This must be set lower than the server's `maximum-time-between-pings`.	`10`
`redundancy-monitor-interval`	The interval, in seconds, at which the subscription HA maintenance thread checks for the configured redundancy of subscription servers.	`10`
`stacktrace-enabled`	If `true`, the exception classes capture a stack trace that can be printed with their `printStackTrace` function. If `false`, the function prints a message that the trace is unavailable.	`false`
	`Licensing Properties`	
`license-file`	The name and path of the file containing the license for the distributed system member, which is read in during connection to the distributed system. See *Licensing* on page 30.	*`productDir`*`/gfCppLicense.zip`
`license-type`	The type of license used by the distributed system member: evaluation, development, or production. All applications in the distributed system must have the same type of license. See *Licensing* on page 30.	`evaluation`
	`Logging Properties`	
`log-disk-space-limit`	The maximum amount of disk space, in megabytes, allowed for all log files, current, and rolled. If set to `0`, the space is unlimited.	`0`
`log-file`	The name and full path of the file where a running client writes log messages. If not specified, logging goes to `stdout`.	no default file
`log-file-size-limit`	The maximum size, in megabytes, of a single log file. Once this limit is exceeded, a new log file is created and the current log file becomes inactive. If set to `0`, the file size is unlimited.	`0`
`log-level`	Controls the types of messages that are written to the application's log. Levels for general operational use are: `config`, `info`, `warning`, and `error`, where `config` logs the most messages and `error` the least. Setting `log-level` to one of the ordered levels causes all messages of that level and greater severity to be printed. Lowering the `log-level` reduces system resource consumption while still providing some logging information for failure analysis.	`config`

`gfcpp.properties` **Attribute (Continued)**	**Description**	**Default**
	`Statistics Archiving Properties`	
`statistic-sampling-enabled`	Controls whether the process creates a statistic archive file.	`true`
`statistic-archive-file`	The name and full path of the file where a running system member writes archives statistics. If `archive-disk-space-limit` is not set, the native client appends the process ID to the configured file name, like `statArchive-`*PID*`.gfs`. If the space limit is set, the process ID is not appended but each rolled file name is renamed to `statArchive-ID.gfs`, where ID is the rolled number of the file.	`./statArchive.gfs`
`archive-disk-space-limit`	The maximum amount of disk space, in megabytes, allowed for all archive files, current, and rolled. If set to `0`, the space is unlimited.	`0`
`archive-file-size-limit`	The maximum size, in bytes, of a single statistic archive file. Once this limit is exceeded, a new statistic archive file is created and the current archive file becomes inactive. If set to `0`, the file size is unlimited.	`0`
`statistic-sample-rate`	The rate, in seconds, that statistics are sampled. Operating system statistics are updated only when a sample is taken. If statistic archival is enabled, then these samples are written to the archive. Lowering the sample rate for statistics reduces system resource use while still providing some statistics for system tuning and failure analysis. You can view archived statistics with the optional VSD utility.	`1`
`enable-time-statistics`	Enables time-based statistics for the distributed system and caching. For performance reasons, time-based statistics are disabled by default. For more information, see Appendix D, *System Statistics*, on page 301.	`false`
	`Durable Client Properties`	
auto-ready-for-events	Whether a non durable client starts to receive and process subscription events automatically on startup. If set to `false`, event startup is not automatic and you need to call the `Cache::readyForEvents` method after your regions and their listeners are initialized.	`true`
`durable-client-id`	The identifier to specify if we want the client to be durable	`empty`
`durable-timeout`	The time, in seconds, a durable client's subscription is maintained when it is not connected to the server before being dropped	`300`

`gfcpp.properties` **Attribute (Continued)**	**Description**	**Default**
	`Security Properties`	
`security-client-dhalgo`	The security diffie-hellman secret key algorithm.	`null`
`security-client-kspath`	The keystore (.pem file) path.	`null`
`security-client-auth-factory`	The factory method for the security `AuthInitialize` module	`empty`
`security-client-auth-library`	The path to the client security library for the `AuthInitialize` module	`empty`

6.3 Defining Properties Programmatically

You can programmatically pass in specific system properties that are customarily defined in `gfcpp.properties`. To do that, create a `Properties` object and define all needed properties individually in that object. See the following example.

Example 6.1 Defining System Properties Programmatically

```
PropertiesPtr systemProps = Properties::create();
systemProps->insert( "statistic-archive-file", "stats.gfs" );
systemProps->insert( "cache-xml-file", "./myapp-cache.xml" );
systemProps->insert( "stacktrace-enabled", "true" );
CacheFactoryPtr systemPtr = CacheFactory::createCacheFactory(systemProps);
```

Chapter 7 Preserving Data

The GemFire Enterprise native client can be configured for data loss prevention in cases where either the cache server that the client is communicating with loses its connection, or if the native client is temporarily disconnected or abnormally terminated from the distributed system.

Data preservation is accomplished by ensuring that reliable event messaging is maintained between the client and the cache server. Any lost messages intended to be sent to a client as `create`, `update` or `invalidate` events for cached entries can quickly cause the data in the client cache to be unsynchronized and out of date with the rest of the distributed system.

This chapter describes the configuration and operation of native clients to prevent data loss in the event of client or server failure.

In this chapter:

7.1 High Availability for Client-to-Server Communication

The GemFire Enterprise native client provides reliable event messaging from cache server to client to prevent data loss during server failover operations. High availability is implemented in the cache server and is configured in the native client. High availability functions the same whether the region is partitioned or not. See the *GemFire Enterprise Developer's Guide* for information about partitioned regions.

Native clients can specify the number of secondary servers where the client registers interest and maintains subscription channels, in addition to the subscription channel with the primary server. The secondary servers maintain redundant update queues for the client. If the primary server fails, one of the secondaries steps in as primary to provide uninterrupted messaging to the client. If possible, another secondary is then initialized so the total number of secondaries is not reduced by the failover.

See the *GemFire Enterprise Developer's Guide* for details about configuring a Java cache server for high availability.

Configuring Native Clients for High Availability

Configure high availability by setting the pool attribute `subscription-redundancy` to the number of copies you want maintained.

A client maintains its queue redundancy level at the time of a primary server failure by connecting to additional secondary servers. This example sets one redundant server as failover backup to the primary server:

Example 7.1 Setting the Server Redundancy Level in cache.xml

```
<cache>
   <pool name="examplePool"
      subscription-enabled="true" subscription-redundancy="1">
      <server host="java_servername1" port="java_port1" />
      <server host="java_servername2" port="java_port2" />
   </pool>
   <region name = "ThinClientRegion1" >
      <region-attributes refid="CACHING_PROXY" pool-name="examplePool"/>
   </region>
</cache>
```

You can set the redundancy level programmatically. This example creates a client cache with two redundant cache servers configured in addition to the primary server.

The server redundancy level can be configured using the pool API. For more information about the pool API, see *Using Connection Pools* on page 235.

Example 7.2 Setting the Server Redundancy Level Programmatically

```
PropertiesPtr pp = Properties::create( );
systemPtr = CacheFactory::createCacheFactory(pp);
// Create a cache.
cachePtr = systemPtr->setSubscriptionEnabled(true)
   ->addServer("localhost", 24680)
   ->addServer("localhost", 24681)
   ->addServer("localhost", 24682)
   ->setSubscriptionRedundancy(2)
   ->create();
```

When failover occurs to a secondary server, a new secondary is added to the redundancy set. If no new secondary server is found, then the redundancy level is not satisfied but the failover procedure completes successfully. Any new live server is added as a secondary and interest is registered on it.

Sending Periodic Acknowledgement

When redundancy is enabled for high availability and `redundancy-level` is set to `1` or higher, clients send (and servers expect) periodic acknowledgement messages at configurable intervals for notifications they have received. A periodic `ack` is not sent by the client if there are no unacknowledged notifications at the time. Servers use this periodic acknowledgement to help reduce the likelihood of sending duplicate notifications, and to reduce resource usage.

The following system properties in the `gfcpp.properties` file are used to configure periodic ack:

`notify-ack-interval`	Sets the minimum period between two consecutive acknowledgement messages sent from the client to the server. The default setting (in seconds) is `10`.
`notify-dupcheck-life`	Sets the minimum time a client continues to track a notification source for duplicates when no new notifications arrive before expiring it. The default setting (in seconds) is `300`.

The Pool API also provides attributes to configure periodic ack and duplicate message tracking timeout. See `subscription-message-tracking-timeout` and `subscription-ack-interval` in the list of pool attributes at *Configuring Pools for Servers or Locators* on page 238.

Using Queue Conflation to Improve Update Performance

The client queues on the cache server can be configured to conflate queued messages so the native client receives only the latest update for a particular entry key. Queue conflation is performed when an entry update is added to the queue. If the last operation queued for that key is also an `update` operation, the previously enqueued update is removed, leaving only the latest update to be sent to the client when event distribution occurs. For high availability, the conflation process is also performed on any secondary queues.

Only entry `update` messages in a cache server region with `distributed-no-ack` scope are conflated. Region operations and entry operations other than updates are not conflated.

Conflation is enabled in the cache server region, so all clients interested in updates in a particular region either get the updates conflated or not. To enable conflation, set the cache server's `enable-bridge-conflation` region attribute to `true`. Conflation is set to `false` by default.

See the queue conflation discussion in the *GemFire Enterprise Developer's Guide* for more information about using conflation.

Per-Client Queue Conflation

Conflation can be overridden on a per-client basis by setting the `conflate-events` property in the native client's `gfcpp.properties` file. Valid settings are:

- `server`—Uses the server settings
- `true`—Conflates everything sent to the client
- `false`—Does not conflate anything sent to the client

7.2 Durable Client Messaging

Clients can configure the redundancy level for their queues that are stored on cache servers. This ensures that the client will not lose messages if it loses the connection to its primary server. Durable messaging allows a disconnected client application to recover its subscribed data when it reconnects to the cache server because the server continues to queue messages for which the client has registered interest.

The messaging queues used for durable messaging are the same regular messaging queues used for basic server-to-client messaging, with all of the requirements, options and functionality described in the *GemFire Enterprise Developer's Guide*. If you are using highly available servers, see *High Availability for Client-to-Server Communication* on page 152 for additional requirements.

For durable messaging, you also need the following:

- **Durable clients**—If the client is durable, the server continues to maintain the client queues when the client disconnects.
- **Durable interest registration**—A durable client's interest registrations specify whether its interest in a key is durable. If it is, the servers continue queuing messages for that key while the client is disconnected.
- **Cache ready message**—When it is ready to receive the stored messages, a durable client must call `Cache.readyForEvents` to send a cache ready message to the server.
- **Disconnect keepalive specification**—When a durable client disconnects normally it must tell the server whether to maintain the message queue or delete it.
- **Durable client callback method**—If you use cache listeners on the durable clients, you have the option to implement the `afterRegionLive` callback method. This callback is invoked after the durable client connects to its servers, when it has received all of its stored messages and replayed the events.

Client-Side Configuration

This section describes the settings for configuring and implementing durable messaging for C++ and .NET clients. All durable messaging configurations are performed on the client.

Configuring a Durable Native Client

The durable native client can be configured in the `gfcpp.properties` file, or in the `DistributedSystem::connect` call.

- **Durable client ID**—You indicate that the client is durable by giving it a `durable-client-ID`. The servers use this ID to identify the client. For a non-durable client, the `durable-client-ID` is an empty string. The ID can be any number that is unique among the clients attached to servers in the same distributed system.
- **Durable timeout**—The `durable-timeout` setting specifies how long this client's servers should wait after the client disconnects before terminating its message queue. During that time, the servers consider the client alive and continue to accumulate messages for it. The default is 300 seconds.

The `durable-timeout` setting is a tuning parameter. When setting the timeout, take into account the normal activity of your application, the average size of your messages, and the level of risk you can handle. Assuming that no messages are being removed from the queue, how long can the application run before the queue reaches the maximum message count? In addition, how long can it run before the queued messages consume all the memory on the client host? How serious is each of those failures to your operation?

To assist with tuning, GemFire Enterprise provides statistics that track message queues for durable clients through the disconnect and reconnect cycles. The statistics are documented in the *GemFire Enterprise System Administrator's Guide*.

When the queue is full it blocks further operations that add messages until the queue size drops to an acceptable level. The action to take is specified on the server. For details on configuring the queue, see the *GemFire Enterprise Developer's Guide*.

The following example shows `gfcpp.properties` settings to make the client durable and set the durable timeout to 200 seconds.

Example 7.3 Configuring a Durable Native Client Using gfcpp.properties

```
durable-client-id=31
durable-timeout=200
```

This programmatic example creates a durable client using the `DistributedSystem::connect` call.

Example 7.4 Configuring a Durable Client Through the API (C++)

```
// Create durable client's properties using the C++ api
PropertiesPtr pp = Properties::create();
pp->insert("durable-client-id", "DurableClientId");
pp->insert("durable-timeout", 200);
cacheFactoryPtr = CacheFactory::createCacheFactory(pp);
```

Configuring Durable Interest in Keys

When a durable client disconnects, its servers keep queuing messages for selected keys. The client indicates which keys by registering durable interest for those keys. This fine-grained control handles the constraints of queue size and memory by saving only the critical messages.

You still register interest for other keys, but not durable interest. When the client is connected to its servers, it receives messages for those non-durable keys. When the client is disconnected, its non-durable interest registrations are deleted but messages that are already in the queue remain there.

For durable clients, all interest registration is done immediately after the regions are created. This is required whether interest registration is durable or not durable. An extra `registerInterest` parameter specified for durable clients indicates whether the registration is durable (`true`) or not (`false`).

The following programmatic example registers durable interest in `Key-1`. The interest registration happens immediately after region creation and before anything else.

Example 7.5 API Client Durable Interest List Registration (C++)

```
// Durable client interest registration can be
// durable (true) or nondurable(default).
VectorOfCacheableKey keys;
keys.push_back( CacheableString::create("Key-1") );
regionPtr->registerKeys(keys,true);
```

You use the typical methods for interest registration and configure notification by subscription on the server as usual. For details, see *Registering Interest for Entries* on page 46.

Changing interest registration after the durable client connects the first time can cause data inconsistency and is not recommended.

At restart, if the client doesn't register durable interest for exactly the same keys as before then the entries in the interest list are not copied from the server during the registration. Instead, the client cache starts out empty and entries are added during updates. If no updates come in for an entry, it never shows up in the client cache.

Durable Client Operations

This section covers the `Cache` interface methods and parameters that are specific to durable clients.

Sending the Cache Ready Message to the Server

After a durable client connects and initializes its cache, regions, and any cache listeners, it invokes `readyForEvents` to indicate to the servers that the client is ready to receive any messages accumulated for it. The following example shows how to call `readyForEvents`.

Example 7.6 Durable Client Cache Ready Notification (C++)

```
//Send ready for event message to server(only for durable clients).
//Server will send queued events to client after receiving this.
cachePtr->readyForEvents();
```

To keep the client from losing events, do not call this method until all regions and listeners are created. For more information, see *Reconnection* on page 158.

Disconnecting From the Server

When a durable client closes its cache and disconnects, it tells the servers whether to maintain its queues. For this purpose, use the version of `Cache::close` with the boolean `keepalive` parameter set, as shown in the following example. If the setting is `true`, the servers keep the durable client's queues and durable subscriptions alive for the timeout period. In addition to in-memory queue retention, the servers can evict the most recent durable client queue updates to disk to reduce memory consumption.

Only the resources and data related to the session are removed, such as port numbers and non-durable subscriptions. If the setting is `false`, the servers do the same cleanup that they would do for a non-durable client.

Example 7.7 Durable Client Disconnect With Queues Maintained

```
// Close the Cache and disconnect with keepalive=true.
// Server will queue events for durable registrations and CQs
// When the client reconnects (within a timeout period) and sends
// "readyForEvents()", the server will deliver all stored events
cachePtr->close(true);
```

Life Cycle of a Durable Client

This section discusses the high-level operation of a durable client through initial startup, disconnection, and reconnection.

Initial Operation

The initial startup of a durable client is similar to the startup of any other client, except that it specifically calls the `Cache.readyForEvents` method when all regions and listeners on the client are ready to process messages from the server. See *Sending the Cache Ready Message to the Server* on page 156.

Disconnection

While the client and servers are disconnected, their operation varies depending on the circumstances.

Normal disconnect

When a durable client disconnects normally, the `Cache.close` request states whether to maintain the client's message queue and durable subscriptions. The servers stop sending messages to the client and release its connection. See *Disconnecting From the Server* on page 156 for more information.

If requested, the servers maintain the queues and durable interest list until the client reconnects or times out. The non-durable interest list is discarded. The servers continue to queue up incoming messages for entries on the durable interest list. All messages that were in the queue when the client disconnected remain in the queue, including messages for entries on the non-durable list.

If the client requests to not have its subscriptions maintained, or if there are no durable subscriptions, the servers unregister the client and perform the same cleanup as for a non-durable client.

Abnormal disconnect

If the client crashes or loses its connections to all servers, the servers automatically maintain its message queue and durable subscriptions until the client reconnects or times out.

Client disconnected but operational

If the client operates while it is disconnected, it gets what data it can from the local cache. Since updates are not allowed, the data can become stale. An `UnconnectedException` occurs if an update is attempted.

Timing out while disconnected

The servers track how long to keep a durable client queue alive based on the `durable-client-timeout` setting. If the client remains disconnected longer than the timeout, the servers unregister the client and do the same cleanup that is performed for a non-durable client. The servers also log an alert. When a timed-out client reconnects, the servers treat it as a new client making its initial connection.

Reconnection

During initialization, operations on the client cache can come from multiple sources:

- Cache operations by the application
- Results returned by the cache server in response to the client's interest registrations
- Callbacks triggered by replaying old events from the queue

These procedures can act on the cache concurrently, and the cache is never blocked from doing operations.

GemFire Enterprise handles the conflicts between the application and interest registration, but you need to prevent the callback problem. Writing callback methods that do cache operations is never recommended, but it is a particularly bad idea for durable clients, as explained in *Implementing Cache Listeners for Durable Clients* on page 161.

When the durable client reconnects, it performs these steps:

1. The client creates its cache and regions. This ensures that all cache listeners are ready. At this point, the application hosting the client can begin cache operations.
2. The client calls `Cache.readyForEvents`, meaning that all regions and listeners on the client are now ready to process messages from the server. The cache ready message triggers the queued message replay process on the primary server.
3. The client issues its register interest requests. This allows the client cache to be populated with the initial interest registration results. The primary server responds with the current state of those entries if they still exist in the server's cache.

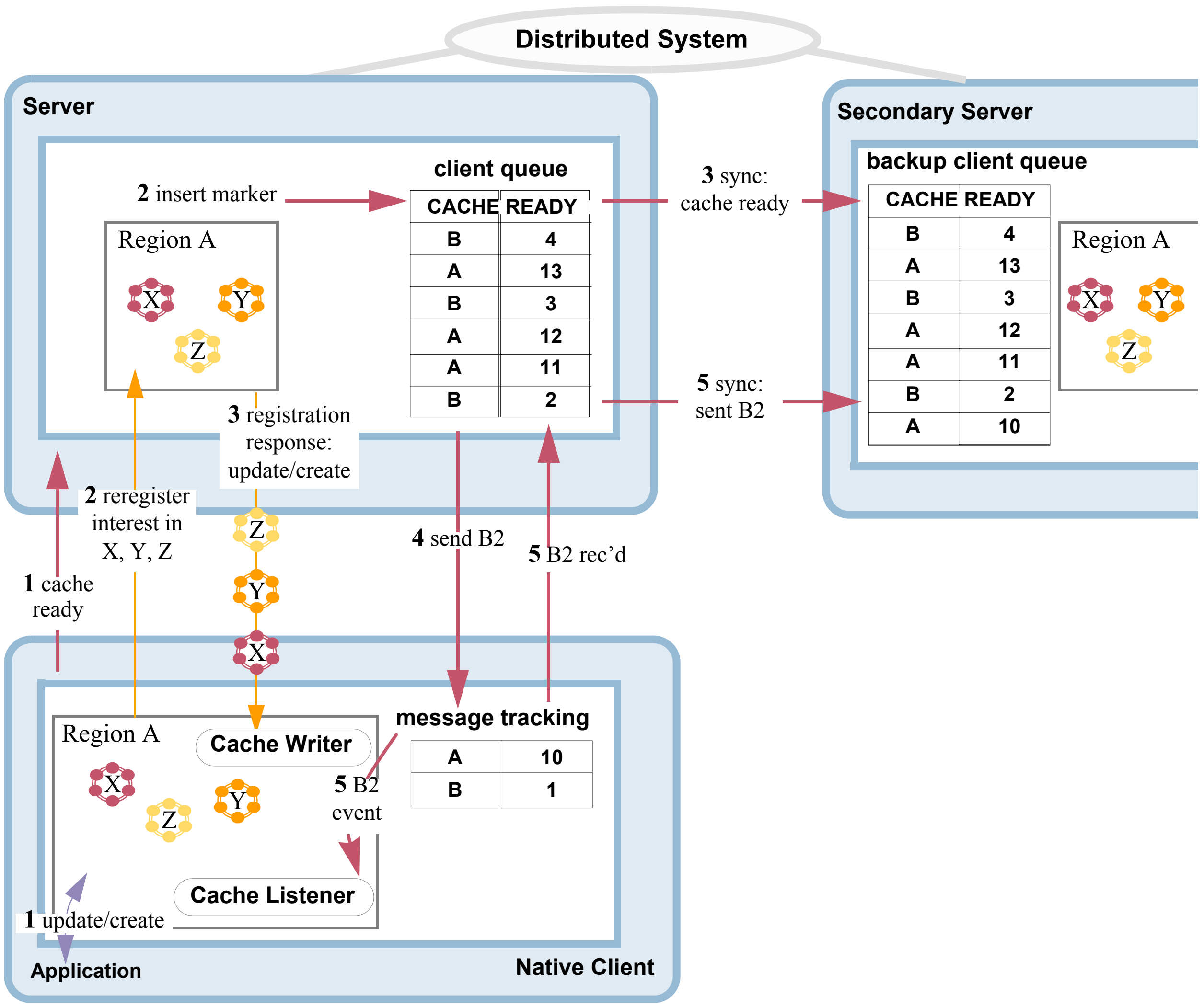
   Interest registration must happen immediately after the cache ready message.

For an example that demonstrates `Cache.readyForEvents`, see *Sending the Cache Ready Message to the Server* on page 156.

This figure shows the concurrent procedures that occur during the initialization process. The application begins operations immediately on the client (step 1), while the client's cache ready message (also step 1) triggers a series of queue operations on the cache servers (starting with step 2 on the primary server). At the same time, the client registers interest (step 2 on the client) and receives a response from the server.

Message B2 applies to an entry in Region A, so the cache listener handles B2's event. Because B2 comes before the marker, the client does not apply the update to the cache.

Figure 7.1 Initialization of a Reconnected Durable Client



Only one region is shown for simplicity, but the messages in the queue could apply to multiple regions. Also, the figure omits the concurrent cache updates on the servers, which would normally be adding more messages to the client's message queue.

Durable Message Replay

When the primary server receives the cache ready message, the servers and client execute the following procedure to update the queue and replay the events from the stored messages. To avoid overwriting current entries with old data, the client does not apply the updates to its cache.

1. The server finds the queue for this durable client ID and updates its information, including the client's socket and remote ports.

   If the client has timed out while it was disconnected, its queues are gone and the server then treats it as a new client. See *Initial Operation* on page 157.

2. All servers that have a queue for this client place a marker in the queue.

   Messages in the queue before the marker are considered to have come while the client was disconnected. Messages after the marker are handled normally.

3. The cache server sends the queued messages to the client. This includes any messages that were evicted to disk.

4. The client receives the messages but does not apply the updates to its cache. If cache listeners are installed, they handle the events. For implications, see *Implementing Cache Listeners for Durable Clients* on page 161.

5. The client receives the marker message indicating that all past events have been played back.

6. The cache server sends the current list of live regions.

7. In each live region on the client, the marker event triggers the `afterRegionLive` callback.

   After the callback, the client begins normal processing of events from the server and applies the updates to its cache.

Even when a new client starts up for the first time, the cache ready markers are inserted in the queues. If messages start coming into the new queues before the servers insert the marker, those messages are considered as having happened while the client was disconnected, and their events are replayed the same as in the reconnect case.

Application Operations During Interest Registration

As soon as the client creates its regions, the application hosting the client can start cache operations, even while the client is still receiving its interest registration responses. In that case, application operations take precedence over interest registration responses.

When adding register interest responses to the cache, the following rules are applied:

- If the entry already exists in the cache with a valid value, it is not updated.
- If the entry is invalid and the register interest response is valid, the valid value is put into the cache.
- If an entry is marked destroyed, it is not updated. Destroyed entries are removed from the system after the register interest response is completed.

If the interest response does not contain any results because all of those keys are absent from the server's cache, the client's cache can start out empty. If the queue contains old messages related to those keys, the events are still replayed in the client's cache.

Implementing Cache Listeners for Durable Clients

Designing a cache listener for durable clients has two unique aspects. First, you must implement all the callback methods to behave properly when stored events are replayed. Second, a cache listener has a callback method, `afterRegionLive`, specifically for durable clients. For general instructions on implementing a cache listener, see *CacheListener* on page 62.

Writing Callbacks for Use With Durable Messaging

Durable clients require special attention to cache callbacks generated by the cache listener. During the initialization window when a reconnecting client has a functioning cache but is still receiving the stored messages from the queue, the client can replay events that are long past. These events are not applied to the cache, but they are sent to the cache listener. If the listener's callbacks invoked by these events make changes to the cache, that could conflict with current operations and create data inconsistencies. Consequently, you need to keep your callback implementations lightweight and not do anything in the cache that could produce incorrect results during this window. For details on implementing callbacks for GemFire event handlers, see the *GemFire Enterprise Developer's Guide*.

Implementing the afterRegionLive Method

If you are using cache listeners, you can implement the `afterRegionLive` callback method provided for durable clients. This callback is invoked when the client has received all the old messages that were stored in its queue while it was disconnected. Implementing this method enables you to do application-specific operations when the client has replayed all of these old events.

If you do not wish to use this callback, and your listener is an instance of `CacheListener` (not a `CacheListenerAdapter`), you must implement `afterRegionLive` as a non-operational method.

Implementation Notes

Redundancy management is handled by the client, so when the client is disconnected from the server the redundancy of client events is not maintained. Even if the servers fail one at a time, so that running clients have time to fail over and pick new secondary servers, an offline durable client cannot fail over. As a result, the client loses its queued messages.

Chapter 8

Security

This chapter describes the security framework for the GemFire Enterprise native client, and explains its authentication and authorization processes with the GemFire distributed system.

The security framework authenticates clients attempting to connect to a GemFire cache server, and authorizes client cache operations. It can also be configured for client authentication of servers. It allows you to plug in your own implementations for authentication and authorization.

In this chapter:

8.1 Authentication

A client is authenticated when it connects to a GemFire cache server that is configured with the client `Authenticator` callback. The connection request must contain valid credentials for the client. Once authenticated, the server assigns the client a unique ID and principal, used to authorize operations. The client must trust all cache servers in the server system as it may connect to any one of them. For information on configuring client/server, see *Standard Client/Server Deployment* on page 201 of the *GemFire Enterprise Developer's Guide*.

GemFire has two types of client authentication:

- Process level— Each pool creates a configured minimum number of connections across the server group. The pool accesses the least loaded server for each cache operation

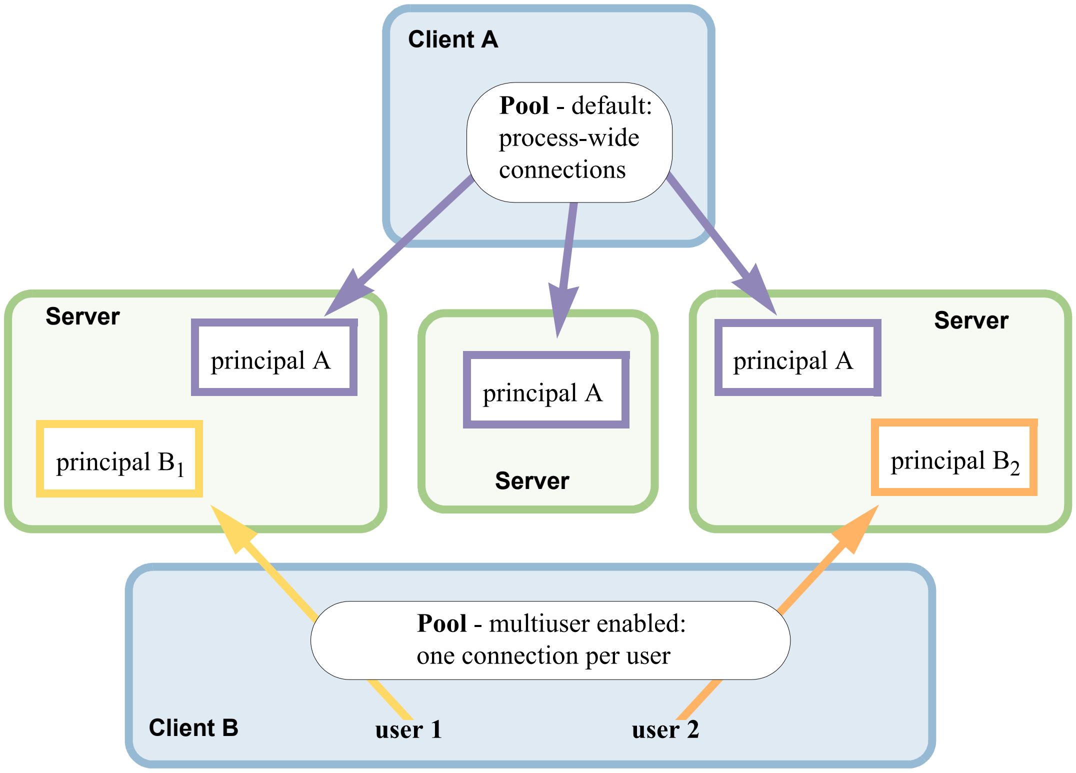
  Process level connections represent the overall client process and are the standard way a client accesses the server cache.

- Multi-user—Each user/pool pair creates a connection to one server and then sticks with it for operations. If the server is unable to respond to a request, the pool selects a new one for the user.

  Multi-user connections are generally used by application servers or web servers that act as clients to GemFire servers. Multi-user allows a single app or web server process to service a large number of users with varied access permissions.

By default, server pools use process level authentication. You can enable multi-user authentication by setting a pool's `multi-user-secure-mode-enabled` attribute to `true`.

Figure 8.1 Client Connections



Credentials can be sent in encrypted form using the Diffie-Hellman key exchange algorithm. See Encrypting Credentials Using Diffie-Hellman (page 173) for more information.

This example shows a C++ client connecting with credentials.

Example 8.1 C++ Client Acquiring Credentials Programmatically

```
PropertiesPtr secProp = Properties::create();
secProp->insert("security-client-auth-factory",
"createPKCSAuthInitInstance");
secProp->insert("security-client-auth-library", "securityImpl");
secProp->insert("security-keystorepath", "keystore/gemfire6.keystore");
secProp->insert("security-alias", "gemfire6");
secProp->insert("security-zkeystorepass", "gemfire");
CacheFactoryPtr cacheFactoryPtr = CacheFactory::createCacheFactory(secProp);
```

This example shows a .NET client.

Example 8.2 .NET Client Acquiring Credentials Programmatically

```
Properties secProp = Properties.Create();
secProp.Insert("security-client-auth-factory",
"GemStone.GemFire.Templates.Cache.Security.
UserPasswordAuthInit.Create");
secProp.Insert("security-client-auth-library", "securityImpl");

secProp.Insert("security-username"," gemfire6");
secProp.Insert("security-password"," gemfire6Pass);
```

System Properties for Authentication

The native client uses system properties to configure authentication. These properties are defined in the `gfcpp.properties` file, which the native client accesses during startup.

security-client-auth-factory

This is the system property for the factory function of the class implementing the `AuthInitialize` interface (`IAuthInitialize` in .NET). The .NET clients can load both C++ and .NET implementations. For .NET implementations, this property is the fully qualified name of the static factory function (including the namespace and class).

security-client-auth-library

This is the system property for the library where the factory methods reside. The library is loaded explicitly and the factory functions are invoked dynamically, returning an object of the class implementing the `AuthInitialize` interface.

Other implementations of the `AuthInitialize` interface may be required to build credentials using properties that are also passed as system properties. These properties also start with the `security-` prefix. For example, the PKCS implementation requires an alias name and the corresponding keystore path, which are specified as `security-alias` and `security-keystorepath`, respectively. Similarly, `UserPasswordAuthInit` requires a username specified in `security-username`, and the corresponding password is specified in the `security-password` system property.

The `getCredentials` function for the `AuthInitialize` interface is called to obtain the credentials. All system properties starting with `security-` are passed to this callback as the first argument to the `getCredentials` function, as shown in the following code snippet:

```
PropertiesPtr getCredentials(PropertiesPtr& securityprops, const char
*server);
```

The following example shows how to implement the factory method in both C++ and .NET:

Example 8.3 Implementing the Factory Method for Authentication (C++ and .NET)

C++ Implementation

```
LIBEXP AuthInitialize* createPKCSAuthInitInstance()
     {
       return new PKCSAuthInit( );
     }
```

.NET Implementation

```
public static IAuthInitialize Create()
{
   return new UserPasswordAuthInit();
}
```

Implementations of the factory method are user-provided. Credentials in the form of properties returned by this function are sent by the client to the server for authentication during the client's handshake process with the server.

The GemFire Enterprise native client installation provides sample security implementations in its `templates/security` folder.

Authentication by the Cache Server

When the cache server receives the client credentials during the handshake operation, the server authenticates the client using the callback configured in the `security-client-authenticator` system property. The handshake succeeds or fails depending on the results of the authentication process.

For example, this is how the `security-client-authenticator` property could be configured in the `gfcpp.properties` file:

```
security-client-authenticator=templates.security.PKCSAuthenticator.create
```

In the previous configuration sample, `PKCSAuthenticator` is the callback class implementing the `Authenticator` interface and `create` is its factory method.

The following example shows an implementation of the static `create` method:

Example 8.4 Implementing the Static create Method

```
public static Authenticator create() {
    return new PKCSAuthenticator();
}
```

For details on the `Authenticator` interface and server side configuration, refer to the *Security* chapter in the *GemFire Enterprise System Administrator's Guide*.

The implementation of `authenticate` *on the server must be compatible with* `getCredentials` *implemented on the client side. The set of properties returned by the* `getCredentials` *method on the client side are sent over to the server side* `authenticate` *method.*

Handling Server Authentication Errors

An `AuthenticatedFailedException` is thrown during region creation or region operations when the credentials presented by a client are rejected by the server.

An `AuthenticationRequiredException` is thrown when the server is configured with security and the client does not present its credentials while attempting to connect. This can occurs if the `security-client-auth-factory` and `security-client-auth-library` properties are not configured on the client.

Creating Multiple Secure User Connections with RegionService

To create multiple, secure connections to your servers from a single client, so the client can service different user types, you create an authenticated `RegionService` for each user.

The most common use case is a GemFire client embedded in an application server that supports data requests from a large number of users. Each user may be authorized to access a subset of data on the servers. For example, customer users may only be allowed to see and update their own orders and shipments.

The authenticated users all access the same `Cache` through instances of the `RegionService` interface. See *RegionService* on page 39.

To implement multiple user connections in your client cache, create your `Cache` as usual, with these additions:

1. Configure your client's server pool for multiple secure user authentication. Example:

   ```
   <pool name="serverPool" multiuser-authentication="true">
      <locator host="host1" port="44444"/>
   </pool>
   ```

   This enables access through the pool for the `RegionService` instances and disables it for the `Cache` instance.

2. After you create your cache, for each user, call your `Cache` instance `createAuthenticatedView` method, providing the user's particular credentials. These are `create` method calls for two users:

   ```
      PropertiesPtr credentials1 = Properties::create();
      credentials1->insert("security-username", "root1");
      credentials1->insert("security-password", "root1");
      RegionServicePtr userCache1 =
         cachePtr->createAuthenticatedView( credentials1 );

      PropertiesPtr credentials2 = Properties::create();
      credentials2->insert("security-username", "root2");
      credentials2->insert("security-password", "root2");
      RegionServicePtr userCache2 =
         cachePtr->createAuthenticatedView( credentials2 );
   ```

   For each user, do all of your caching and region work through the assigned region service pointer. Use the region service to get your regions, and the query service, if you need that, and then do your work with them. Access to the server cache will be governed by the server's configured authorization rules for each individual user.

3. To close your cache, close the `Cache` instance.

Requirements and Caveats for RegionService

For each region, you can perform operations through the `Cache` instance or the `RegionService` instances, but not both.

> *You can use the `Cache` to create a region that uses a pool configured for multi-user authentication, then access and do work on the region using your `RegionService` instances.*

To use `RegionService`:

- Regions must be configured as `EMPTY`. Depending on your data access requirements, this configuration might affect performance, because the client goes to the server for every `get`.
- If you are running durable CQs through the region services, stop and start the offline event storage for the client as a whole. The server manages one queue for the entire client process, so you need to request the stop and start of durable client queue (CQ) event messaging for the cache as a whole, through the `ClientCache` instance. If you closed the `RegionService` instances, event processing would stop, but the events from the server would continue, and would be lost.

  Stop with:

  - `cachePtr->close(true);`

  Start up again in this order:

  a. Create the cache.

  b. Create all region service instances. Initialize CQ listeners.

  c. Call the cache `readyForEvents` method.

Authentication Example with UserPassword and PKCS

This section discusses the concepts and configurations for the sample UserPassword and PKCS implementations. Descriptions of their interfaces, classes and methods are available in the online API documentation.

Disclaimer: These security samples serve only as example implementations. The implementation and its source code is provided on an "as-is" basis, without warranties or conditions of any kind, either express or implied. You can modify these samples to suit your specific requirements and security providers. GemStone Systems, Inc. takes no responsibility and accepts no liability for any damage to computer equipment, companies or personnel that might arise from the use of these samples.

Note that the native client samples are provided in source form only in the "templates" directory within the product directory

.Using an LDAP Server for Client Authentication

An LDAP server can be used by a GemFire cache server using the sample LDAP implementation provided in GemFire server product. See the *Security* chapter in the *GemFire Enterprise System Administrator's Guide* to verify authentication credentials for native clients attempting to connect to the GemFire servers and sending username and passwords using the sample UserPassword scheme.

The username and password with this sample implementation is sent out in plaintext. For better security, either turn on credential encryption using Diffie-Hellman key exchange, or use a scheme like PKCS.

When a client initiates a connection to a cache server, the client submits its credentials to the server and the server submits those credentials to the LDAP server. To be authenticated, the credentials for the client need to match one of the valid entries in the LDAP server. The credentials can consist of the entry name and the corresponding password. If the submitted credentials result in a connection to the LDAP server because the credentials match the appropriate LDAP entries, then the client is authenticated and granted a connection to the server. If the server fails to connect to the LDAP server with the supplied credentials then an AuthenticationFailedException is sent to the client and its connection with the cache server is closed.

Configuration Settings

In the `gfcpp.properties` file for the client, you need to specify the `UserPasswordAuthInit` callback, the username, and the password, like this:

```
security-client-auth-library=securityImpl
security-client-auth-factory=createUserPasswordAuthInitInstance
security-username=<username>
security-password=<password>
```

For server side settings and LDAP server configuration, see the *Security* chapter in the *GemFire Enterprise System Administrator's Guide*.

Using PKCS for Encrypted Authentication

With PKCS, clients send encrypted authentication credentials in the form of standard PKCS signatures to a GemFire cache server when they connect to the server. The credentials consist of the alias name and digital signature created using the private key that is retrieved from the provided keystore. The server uses a corresponding public key to decrypt the credentials. If decryption is successful then the client is authenticated and it connects to the cache server. For unsuccessful decryption, the server sends an `AuthenticationFailedException` to the client, and the client connection to the cache server is closed.

When clients require authentication to connect to a cache server, they use the `PKCSAuthInit` class implementing the `AuthInitialize` interface to obtain their credentials. For the PKCS sample provided by GemFire, the credentials consist of an alias and an encrypted byte array. The private key is obtained from the `PKCS#12` keystore file. To accomplish this, `PKCSAuthInit` gets the alias retrieved from the `security-alias` property, and the keystore path from the `security-keystorepath` property. `PKCSAuthInit` also gets the password for the password-protected keystore file from the `security-keystorepass` property so the keystore can be opened.

To use the PKCS sample implementation, you need to build OpenSSL before building the `securityImpl` library.

Configuration Settings

In the `gfcpp.properties` file for the client, you need to specify the `PKCSAuthInit` callback, the keystore path, the security alias, and the keystore password, like this:

```
security-client-auth-library=securityImpl
security-client-auth-factory=createPKCSAuthInitInstance
security-keystorepath=<PKCS#12 keystore path>
security-alias=<alias>
security-keystorepass=<keystore password>
```

For server side settings, see the description of PKCS sample in the *Security* chapter in the *GemFire Enterprise System Administrator's Guide*.

8.2 Client Authorization

Each server can be configured to authorize some or all cache operations using a provided callback that implements the `AccessControl` interface. The callback can also modify or even disallow the data being provided by the client in the operation, such as a `put` or a `putAll` operation. The callback can also register itself as a post-processing filter that is passed operation results like `get`, `getAll`, and `query`.

Authorization can be done on a per-client basis for various cache operations such as creates, gets, puts, query invalidations, interest registration, and region destroys. On the server side, the `security-client-accessor` system property in the server's `gemfire.properties` file specifies the authorization callback. For example:

```
security-client-accessor=templates.security.XmlAuthorization.create
```

In the previous system property setting, `XmlAuthorization` is the callback class which implements the `AccessControl` interface. The `XmlAuthorization` sample implementation provided with GemFire expects an XML file that defines authorization privileges for the clients. For details of this sample implementation and the `AccessControl` interface, see the *Authorization Using the XmlAuthorization Sample* section in the *Security* chapter for the *GemFire Enterprise System Administrator's Guide*.

Post-Operation Authorization

Authorization in the post-operation phase occurs on the server after the operation is complete and before the results are sent to the client. The callback can modify the results of certain operations, such as `query`, `get` and `keySet`, or even completely disallow the operation. For example, a post-operation callback for a query operation can filter out sensitive data or data that the client should not receive, or even completely fail the operation.

The `security-client-accessor-pp` system property in the server's `gemfire.properties` file specifies the callback to invoke in the post-operation phase. For example:

```
security-client-accessor-pp=templates.security.XmlAuthorization.create
```

Using OperationContext to Determine Pre- or Post-Operation

The `OperationContext` object that is passed to the `authorizeOperation` method of the callback as the second argument provides a `isPostOperation` method which returns `true` when the callback is invoked in the post-operation phase. This is demonstrated in the following example.

Example 8.5 Determining Pre-Operation or Post-Operation Authorization

```
bool authorizeOperation(Region region, OperationContext context)
{
   If(context.isPostOperation())
   {
      //It's a post-operation
   }
   else
   {
     //it's a pre-operation
   }
}
```

If an authorization failure occurs in a pre-operation or post-operation callback on the server, the operation throws a `NotAuthorizedException` on the client.

For more information, see the *Configuring Authorization* section in the *Security* chapter for the *GemFire Enterprise System Administrator's Guide*.

8.3 Encrypting Credentials Using Diffie-Hellman

For secure transmission of sensitive credentials like passwords, you can encrypt credentials using the Diffie-Hellman key exchange algorithm. You do this by setting the `security-client-dhalgo` system property to the name of a valid symmetric key cipher supported by the JDK. Valid `security-client-dhalgo` property values are `DESede`, `AES`, and `Blowfish`, which enable the Diffie-Hellman algorithm with the specified cipher to encrypt the credentials.

For the `AES` and `Blowfish` algorithms, you can also optionally specify the key size for the `security-client-dhalgo` property. Valid key size settings for the `AES` algorithm are `AES:128`, `AES:192`, and `AES:256`. The colon separates the algorithm name and the key size. For the `Blowfish` algorithm, key sizes from 128 to 448 bits are supported.

For `AES` algorithms, you may need Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files from Sun or equivalent for your JDK.

Adding settings for Diffie-Hellman on clients also enables challenge response from server to client in addition to encryption of crendentials using the exchanged key to avoid replay attacks from clients to servers. Clients can also enable authentication of servers, with challenge-response from client to server to avoid server side replay attacks.

Enabling Diffie-Hellman

To encrypt credentials, set `security-client-dhalgo` in the `gfcpp.properties` file to the password for the public key file store on the client. Example:

```
security-client-dhalgo=Blowfish:128
```

This causes the server to authenticate the client using the Diffie-Hellmant algorithm.

Client authentication of server

With Diffie-Hellman enabled, you can also have your client authenticate its servers:

4. Generate a `.pem` file for each pkcs12 keystore:

   4.1 Enter this command from a pkcs12 file or a pkcs keystore:

   ```
   user@host: ~> openssl pkcs12 -nokeys -in <keystore/pkcs12 file> -out
   <outputfilename.pem >
   ```

   4.2 Concatenate the generated `.pem` files into a single `.pem` file. You will use its name in the next step.

1. In the `gfcpp.properties` file

   1.1 Set `security-client-kspath` to the name of the `.pem` file. password for the public key file store on the client.

   1.2 Set `security-client-kspasswd` to the password for the public key file store on the client.

8.4 System Properties

The following table describes the security-related system properties in the `gfcpp.properties` file for native client authentication and authorization.

Table 8.1 System Properties for Client Authentication and Authorization

`security-client-auth-factory`	Sets the key for the `AuthInitialize` factory function.
`security-client-auth-library`	Registers the path to the `securityImpl.dll` library.
`security-client-dhalgo`	Returns the Diffie-Hellman secret key cipher algorithm.
`security-client-kspath`	Specifies the path to a `.pem` file, which contains the public certificates for all GemFire cache servers to which the client can connect through specified endpoints.
`security-client-kspasswd`	Specifies the password for the public key file store on the client.
`security-keystorepath`	Specifies the path to the public keystore.
`security-alias`	Specifies the alias name for the key in the keystore.
`security-keystorepass`	Sets the password for the password-protected keystore.
`ssl-enabled`	Enables SSL-based client/server communication when set to `true`. The default is `false`, which causes communication to use plain socket connections. When this is set to `true`, the other `ssl-*` settings are required.
`ssl-keystore`	Specifies the name of the `.PEM` keystore file, containing the client's private key. Not set by default. Required if `ssl-enabled` is `true`.
`ssl-truststore`	Specifies the name of the `.PEM` truststore file, containing the servers' public certificate. Not set by default. Required is `ssl-enabled` is `true`.

8.5 SSL Client/Server Communication

This section describes the steps required to implement SSL-based communication between your clients and servers.

This assumes you have the necessary SSL libraries. If you do not have them already, download and build copies of the OpenSSL and ACE_SSL third party libraries.

Building the ACE SSL DLL

To download the ACE SSL for build, see http://www.cs.wustl.edu/~schmidt/ACE-install.html.

Unix

1. Make sure the OpenSSL header file directory is in your compiler's `include` path, and that OpenSSL libraries are in your library link/load path, `LD_LIBRARY_PATH`.
2. If you installed OpenSSL into a set of directories unknown by the compiler, set the `SSL_ROOT` environment variable to point to the top level directory of your OpenSSL distribution—the parent directory to OpenSSL's `include` and `lib` directories.
3. Build ACE as described above. When building ACE, add `ssl=1` to your `make` command line invocation, or add it to your `platform_macros.GNU` file.
4. Set the `ACE_ROOT` environment variable.
5. Build the `ACE_SSL` library in the `$ACE_ROOT/ace/SSL` directory.

Microsoft Visual Studio

1. Set the `SSL_ROOT` environment variable to the location of the directory containing the OpenSSL `inc32` and `out32dll` directories.
2. Add `ssl=1` to your MPC `$ACE_ROOT/bin/MakeProjectCreator/config/default.features` or `$ACE_ROOT/local.features` file, and re-run MPC to add support for building the `ACE_SSL` library to your MSVC++ workspaces and projects.
3. Open the `ACE.sln` solution, and refer to the ACE build and installation instructions above for details on creating a `config.h` configuration header for this platform.
4. Once the `config.h` file has been created, build the `ACE_SSL` project.

Building the OpenSSL DLL

Compile the OpenSSL DLL. See *Using OpenSSL* on page 177.

Key and keystore management

The GemFire server requires keys and keystores in the Java Key Store (JKS) format while the native client requires them in the clear PEM format. Because of this, you need to be able to generate private/public keypairs in either format and convert between the two.

There are public third party free tools and source code available to download such as from http://yellowcat1.free.fr/keytool_iui.html.

System properties

1. set ssl-enabled (page 174) to `true`
2. set ssl-keystore (page 174) and ssl-truststore (page 174) to point to your files

Starting and stopping the client and server

Configure the native client with the ssl properties as described and with the servers or locators specified as usual.

1. Make sure the OpenSSL and ACE_SSL `DLL`s are built and their locations are in the right environment variables for your system: `PATH` for Windows, and `LD_LIBRARY_PATH` for Unix.
2. Generate the keys and keystores and set the system properties.
3. Set these SSL properties in the locator's `gemfire.properties` file:

```
ssl-enabled=true
ssl-protocols=any
ssl-require-authentication=true
ssl-ciphers=SSL_RSA_WITH_NULL_SHA
```

See the *GemFire System Administrator's Guide* for details on stopping and starting locators and cache servers with SSL.

Example locator start command:

```
gemfire start-locator -port=12345 -dir=. -Djavax.net.ssl.keyStore=
gemfire.jks -Djavax.net.ssl.keyStorePassword=gemfire
-Djavax.net.ssl.trustStore=gemfire.jks
-Djavax.net.ssl.trustStorePassword=gemfire
```

Example locator stop command:

```
gemfire stop-locator -port=12345 -dir=. -Djavax.net.ssl.keyStore=
gemfire.jks -Djavax.net.ssl.keyStorePassword=gemfire
-Djavax.net.ssl.trustStore=gemfire.jks
-Djavax.net.ssl.trustStorePassword=gemfire
```

Example server start comand:

```
cacheserver start -J-Xmx1024m -J-Xms128m locators=host:12345
cache-xml-file=server.xml -dir=. mcast-port=0 log-level=fine ssl-enabled=
true ssl-require-authentication=true ssl-ciphers=SSL_RSA_WITH_NULL_SHA
-J-Djavax.net.ssl.keyStore=gemfire.jks
-J-Djavax.net.ssl.keyStorePassword=gemfire -J-Djavax.net.ssl.trustStore=
gemfire.jks  -J-Djavax.net.ssl.trustStorePassword=gemfire
security-log-level=fine
```

Example server stop command:

```
cacheserver stop -dir=.
```

Limitations

The keys and keystores need to be in the JKS (Java KeyStore) format for the GemFire server and in the clear PEM format for the Native Client.

Currently the native client only support the `NULL` cipher with mutual authentication for SSL socket communications.

8.6 Using OpenSSL

The open-source OpenSSL toolkit provides a full-strength general purpose cryptography library to operate along with the PKCS sample implementation for encrypted authentication of native client credentials.

This section provides information about how to configure your environment to run OpenSSL, where to download the OpenSSL source files, and how to build or install OpenSSL.

If you use Cygwin, it is recommended that you do not use the OpenSSL library that comes with Cygwin because it is built with `cygwin.dll` *as a dependency.*

Configuring the Environment

This section lists the commands for configuring your system environment to build and run OpenSSL. Follow the environment setup that applies to your operating system. For all references to the `NativeClient_xxxx` directory, replace the **xxxx** with the actual four-digit product number.

Bourne and Korn shells (sh, ksh, bash)

```
% OPENSSL=<parent folder for OpenSSL binaries>; export OPENSSL
% GFCPP=<path to installation, typically C:\NativeClient_xxxx>; export GFCPP
% LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$GFCPP/lib:$OPENSSL/lib
% export LD_LIBRARY_PATH
% CLASSPATH=$GEMFIRE/lib/gfSecurityImpl.jar:$CLASSPATH
```

Windows

```
> set GFCPP=<path to installation, typically C:\NativeClient_xxxx>
> set OPENSSL=<path to installed OpenSSL, typically C:\OpenSSL>
> set PATH=<path to Java JDK or JRE>\bin;%GFCPP%\bin;%OPENSSL%\bin;%PATH%
> set CLASSPATH=<path to GemFire installation>\lib\gfSecurityImpl.jar;%CLASSPATH%
```

Downloading OpenSSL

This section lists the web sites where you can download OpenSSL, and additional packages for each supported platform. For Linux and Solaris, you can download the latest tarball archive from the OpenSSL web site at http://www.openssl.org/source/.

Linux

OpenSSL is installed on Linux by default, but the OpenSSL development package may need to be installed to successfully compile the security templates.

Solaris

For Solaris, you can also download a pre-built version of OpenSSL from the Sunfreeware site: See *Pre-Built OpenSSL* on page 178 for more information.

Windows

An OpenSSL installer can be downloaded for Windows from http://www.openssl.org/related/binaries.html

Building or Installing OpenSSL

Follow these instructions to build or install OpenSSL for your specific operating system.

Linux

To build OpenSSL on Linux, copy the downloaded tarball file into `NativeClient_xxxx/templates/security/openssl/Linux` and run `buildit.sh`.

Solaris

To build OpenSSL on Solaris, copy the downloaded tarball file into `NativeClient_xxxx/templates/security/openssl/SunOS` and run `buildit.sh`.

Pre-Built OpenSSL

If you do not want to build OpenSSL, you can download a pre-built archive of OpenSSL for Solaris at http://www.sunfreeware.com/. Follow the links for your specific Solaris operating system to download a compatible version of OpenSSL and install it. To complete the installation, you may need to install some additional listed support files.

Windows

Use the downloaded OpenSSL installer to install it on Windows. You can usually accept the default installation path (`C:\OpenSSL`).

Chapter 9 Remote Querying

You can use the GemFire Enterprise native client query API to query your cached data stored on a GemFire Enterprise cache server. The query is evaluated and executed on the cache server, then the results are returned to the native client. You can also optimize your queries by defining indexes on the cache server.

The query language for the native client is essentially a subset of OQL (ODMG 3.0 Object Data Management Group, www.odmg.org). OQL is an SQL-like language with extended functionality for querying complex objects, object attributes and methods.

In this chapter:

The online C++ and .NET API documentation located in the `docs` directory for the native client provides extensive details for all of the querying interfaces, classes and methods.

It is assumed that you have general familiarity with SQL querying and indexing, and with the information on the native client cache provided in the previous chapters of this manual.

If you are using the new pool API, you should obtain the `QueryService` from the `pool`. For information about the pool API, see *Using Connection Pools* on page 235.

9.1 Quick Introduction to Remote Querying

This discussion provides a quick look at native client remote querying to the GemFire Enterprise cache server. It provides querying examples and shows how the APIs are used to run queries against cached data. It also introduces the data that is used in the examples.

Executing a Query

To execute a query from the native client:

1. Get a pointer to the `QueryService`. Example:

   ```
   QueryServicePtr qsPtr = cachePtr->getQueryService();
   ```

2. Create a `QueryPtr` to a query that is compatible with the OQL specification. Example:

   ```
   QueryPtr qry = qsPtr->newQuery(
      "select distinct * from /Portfolios where status = 'active'");
   ```

3. Use the `execute` method for the `Query` interface to submit the query string to the cache server.

   ```
   SelectResultsPtr results = qry->execute(10);
   ```

   The server remotely evaluates the query string and returns the results to the client.

4. You can iterate through the returned objects as part of the query process. Example:

   ```
   SelectResultsIterator iter = results->getIterator();
   while(iter.hasNext())
   {
   PortfolioPtr portfolio = dynCast<PortfolioPtr >(iter.next());
   }
   ```

The examples in this chapter all assume that you have already obtainted a pointer to the `QueryService`.

The Query Example Data Used in This Chapter

All queries that use this example data assume that the `/portfolios` region has `javaobject.Portfolio` objects on the cache server.

Requirements for Remote Querying

Remote querying for the native client has the following requirements:

- When you are using region endpoints, at least one region must exist on the native client before a query can be executed through the client.
- `SELECT` must always be followed by `DISTINCT` in query strings.
- All objects in the region belong to the same class hierarchy (homogeneous types).

Setting Server Region Data Policy and Scope

Native client remote querying only accesses the data that is available in the remote cache server region, so no local cache loading operations are performed. Depending on the cache server region's `scope` and `data-policy` attribute settings, this could mean that your queries and indexes only see a part of the data available for the server region in the distributed cache.

To ensure a complete data set for your queries and indexes, your cache server region must have its data policy set to `replicate` *or* `persistent-replicate`.

For a cache server region, setting its data policy to `replicate` or `persistent-replicate` ensures that it reflects the state of the entire distributed region. Without replication, some server cache entries may not be available.

Depending on your use of the server cache, the non-global distributed scopes `distributed-ack` and `distributed-no-ack` may encounter race conditions during entry distribution that cause the data set to be out of sync with the distributed region. The `global` scope guarantees data consistency across the distributed system, but at the cost of reduced performance.

Table 9.1 on page 181 summarizes the effects of cache server region scope and data policy settings on the data available to your querying and indexing operations.

Table 9.1 The Effects of Cache Server Region Scope and Data Policy on the Data Available for Querying

Region type Scope	Not replicated	Replicated
`distributed-ack` or `distributed-no-ack`	N/A	FULL data set (if no race conditions)
`global`	N/A	FULL data set

For more information, see the *Scope* and *Data Policy* discussions in the *GemFire Enterprise Developer's Guid*e.

Implementing the equals and hashCodes methods

The `Portfolio` and `Position` query objects for the cache server must have the `equals` and `hashCode` methods implemented, and those methods must provide the properties and behavior mentioned in the online documentation for `Object.equals` and `Object.hashCode`. Inconsistent query results can occur if these methods are absent.

See the `Object` class description in the GemFire Enterprise online Java API documentation for more information about the `equals` and `hashCode` methods.

Setting Object Type Constraints

Performing queries on cache server regions containing heterogeneous objects, which are objects of different data types, may produce undesirable results. Queries should be performed only on regions that contain homogeneous objects of the same object type, although subtypes are allowed. See *The IN Expression* on page 206 for a subtype example.

So your queries will address homogeneous data types, you need to be aware of the values that the client adds to the server. You can set the `key-constraint` and `value-constraint` region attributes to restrict region entry keys and values to a specific object type. However, since objects put from the client remain in serialized form in the server cache and don't get deserialized until a query is executed, it is still possible to put heterogeneous objects from the client.

See the *GemFire Enterprise Developer's Guide* for descriptions of the `key-constraint` and `value-constraint` attributes for the cache server. See *Specifying the object types of FROM clause collections* on page 197 for more information on associating object types with queries.

The Example portfolios Region

Most of the examples in this discussion use the `/portfolios` region, whose keys are the portfolio ID and whose values contain the summarized data shown in the following example.

Example 9.1 C++ Class Definition and Corresponding Java Class Definition

Sample C++ class definition

```
class Portfolio : public Serializable {
   int ID;
   char * type;
   char * status;
   Map<Position> positions;
}
class Position : public Serializable {
   char * secId;
   double mktValue;
   double qty;
}
```

Corresponding Java class definition

```
class Portfolio implements DataSerializable {
   int ID;
   String type;
   String status;
   Map positions;
}
class Position implements DataSerializable {
   String secId;
   double mktValue;
   double qty;
}
```

User-defined data types must implement the `Serializable` interface on the native client side, while corresponding Java classes must implement the `DataSerializable` interface. The C++ objects for the native client must correspond to the Java objects for the GemFire Enterprise cache server. This means that an object on one side should deserialize correctly to the other side. The following table lists the sample data in the `/portfolios` region:

Table 9.2 Entry values in /portfolios

id	type	status	Positions		
			secId	mktValue	qty
111	xyz	active	xxx	27.34	1000.00
			xxy	26.31	1200.00
			xxz	24.30	1500.00
222	xyz	active	yyy	18.29	5000.00
333	abc	active	aaa	24.30	10.00
			aab	23.10	15.00
444	abc	inactive	bbb	50.41	100.00
			bbc	55.00	90.00

Querying the portfolios Region

The following examples provide a sampling of the queries that you could run against `/portfolios` on the server. The query results for the data are listed in the previous table. For the first several, the coding examples are included as well to show how you can execute the queries using the API.

Example 9.2 Retrieve all active portfolios

Query string

```
SELECT DISTINCT * FROM /portfolios WHERE status = 'active'
```

Results A collection of `Portfolio` objects for IDs `111`, `222` and `333`.

Code

```
QueryServicePtr qrySvcPtr =
   cachePtr->getQueryService("examplePool");
QueryPtr qry = qrySvcPtr->newQuery(
   "select distinct * from /Portfolios where status = 'active'");
SelectResultsPtr resultsPtr = qry->execute(10);
SelectResultsIterator iter = resultsPtr->getIterator();
while(iter.hasNext())
{
   PortfolioPtr portfolio = dynCast<PortfolioPtr >(iter.next());
}
```

Notes A query response timeout parameter of 10 seconds is specified for the `execute` method to allow sufficient time for the operation to succeed.

Example 9.3 Retrieve all portfolios that are active and have type xyz

Query string

```
SELECT DISTINCT * FROM /portfolios
WHERE status = 'active' AND "type" = 'xyz'
```

Results

A collection of `Portfolio` objects for IDs `111` and `222`

Code

```
QueryServicePtr qrySvcPtr =
   cachePtr->getQueryService("examplePool");
QueryPtr qry = qrySvcPtr->newQuery("select distinct * from
/Portfolios where status = 'active' and \"type\"='xyz'");
SelectResultsPtr results = qry->execute(10);
SelectResultsIterator iter = results->getIterator();
while(iter.hasNext())
{
   PortfolioPtr portfolio = dynCast<PortfolioPtr >(iter.next());
}
```

Notes

The `type` attribute is passed to the query engine in double quotes to distinguish it from the query keyword of the same name.

A query response timeout parameter of 10 seconds is specified for the `execute` method to allow sufficient time for the operation to succeed.

Example 9.4 Get the ID and status of all portfolios with positions in secId ‘yyy’

Query string

```
SELECT DISTINCT id, status FROM /portfolios
WHERE NOT (SELECT DISTINCT * FROM positions.values posnVal TYPE
Position WHERE posnVal.secId='yyy').isEmpty
```

Results

A collection of `Struct` instances, each containing an `id` field and a `status` field. For this data, the collection length is 1 and the `Struct` contains data from the entry with id `222`.

Code

```
QueryServicePtr qrySvcPtr = cachePtr-
>getQueryService("examplePool");
QueryPtr qry = qrySvcPtr->newQuery(
   "import javaobject.Position; SELECT DISTINCT ID, status FROM "
   "/Portfolios WHERE NOT (SELECT DISTINCT * FROM positions.values
"
   "posnVal TYPE Position WHERE posnVal.secId='DELL').isEmpty");
SelectResultsPtr results = qry->execute(10);
SelectResultsIterator iter = results->getIterator();
while(iter.hasNext())
{
   Struct * si = (Struct*) iter.next().ptr();
   SerializablePtr id = si->operator[]("ID");
   SerializablePtr status = si->operator[]("status");
   printf("\nID=%s, status=%s", id->toString()->asChar(), status-
>toString()->asChar());
}
```

Example 9.5 Get distinct positions from all active portfolios with at least a $25.00 market value

Query string

```
SELECT DISTINCT posnVal
FROM /portfolios qryP, qryP.positions.values posnVal TYPE Position
WHERE qryP.status = 'active' AND posnVal.mktValue >= 25.00
```

Results: Collection of `Position` instances with `secId`: `xxx`, `xxy`

Notes: This query assigns iterator variable names to the collections in the `FROM` clause. For example, the variable `qryP` is the iterator for the entry values in the `/portfolios` region. This variable is used in the second part of the `FROM` clause to access the values of the `positions` map for each entry value.

Example 9.6 Get distinct positions from portfolios with at least a $25.00 market value

Query string

```
SELECT DISTINCT posnVal
FROM /portfolios, positions.values posnVal TYPE Position
WHERE posnVal.mktValue >= 25.00
```

Results: Collection of `Position` instances with `secId`: `xxx`, `xxy`, `bbb`, `bbc`

Modifying Cache Contents

The query service is a data access tool, so it does not provide any cache update functionality. To modify the cache based on information retrieved through querying, retrieve the entry keys and use them in the standard entry update methods. For an example of entry key retrieval, see the next example.

Example 9.7 Get distinct entry keys and positions from active portfolios with at least a $25.00 market value

Query string

```
SELECT DISTINCT key, posnVal
FROM /portfolios.entrySet, value.positions.values posnVal TYPE
Position
WHERE posnVal.mktValue >= 25.00
```

Results: A `SelectResults` of `Struct` instances containing `key`, `Position` pairs:

Notes: Retrieving the entry keys allows you to access the cached region entries for update. You cannot update the cache through the query engine.

9.2 Querying Object Data

This discussion shows how to query object data, which is usually nested. To access nested data you must drill down through the outer data layers, so querying nested data also requires a drill-down approach. Data that is not "on the surface" must be brought into scope within a query in order to be accessed. This requires a special approach to querying that is slightly different from classic SQL querying. Object data querying is compared with querying classic relational data, and it provides a graphic view of the changes in data visibility as a GemFire cache query executes.

Querying Object Data Versus Relational Data

The query package allows you to access cached regions and region data with SQL-style querying. If you have ever queried database tables using SQL, many of the query package concepts will be familiar to you. There are differences between the relational database table storage model and GemFire's object storage model that translate into subtle but important differences in querying techniques. This discussion provides a comparison of the two storage models and their querying techniques.

The comparison uses the portfolios example from the prior section. Each portfolio has an ID, a status, a type, and a map of associated positions. Each position represents a `secId`, a `mktValue`, and a `qty`. In a database you would store this information in a portfolio table and a position table, connecting the two tables through primary and foreign keys. In a cache region, the position data would be stored as a map inside the portfolio data. The following figures shows the data layouts followed by the models with some of the data from Table 9.2 on page 183:

Figure 9.1 Storage Model Comparison: Database and Cache

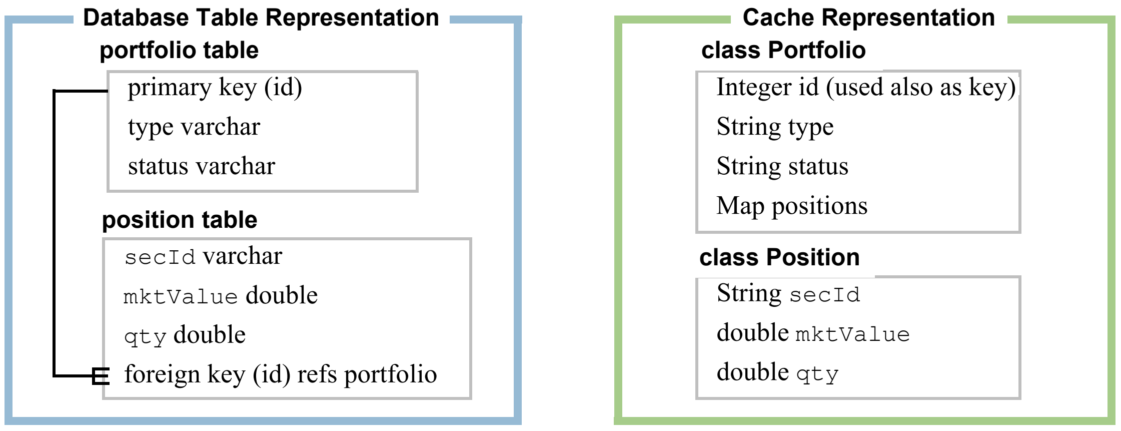
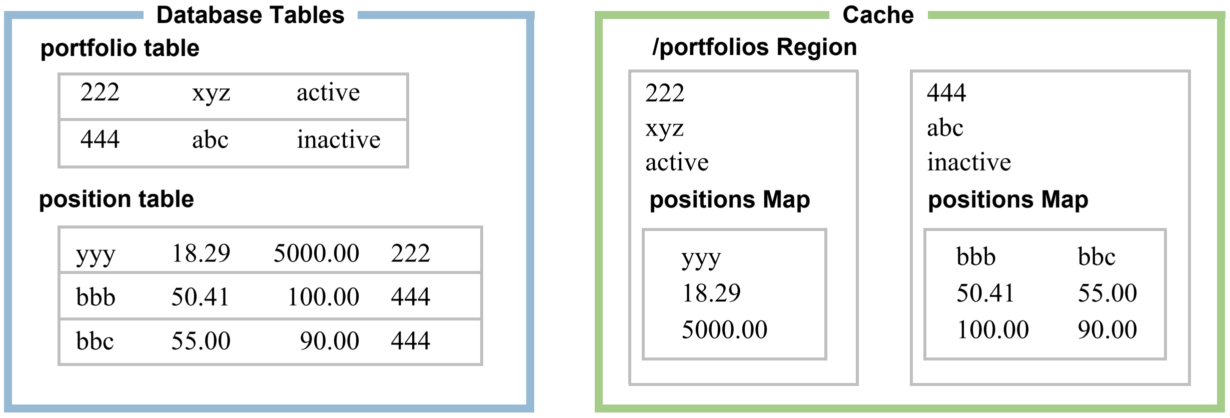


Figure 9.2 Database and Cache Storage Models With Data



As you can see, the classic database storage model is flatter, with each table standing on its own. You can access position data independent of the portfolio data, or you can join the two tables if you need to by using multiple expressions in your `FROM` clause. With the object model, on the other hand, you use multiple expressions in the `FROM` clause to drill down into the portfolio table and access the position data. This can be helpful, since the position-to-portfolio relationship is implicit and does not require restating in your queries. It may also hinder you when you want to ask general questions about the positions data independent of the portfolio information. The two queries below illustrate these differences.

You want to list the market values for all positions of active portfolios. You query the two data models with these statements:

Database Table Query 1	Cache Query 1

`SELECT mktValue`
`      FROM portfolio, position`
`      WHERE status = "active"`
`      AND portfolio.id =`
`position.id`
	`IMPORT cacheRunner.Position;`
	`SELECT DISTINCT posnVal.mktValue`
	`   FROM /portfolios,`
	`      positions.values posnVal TYPE Position`
	`   WHERE status='active'`

The difference between the queries reflects the difference in data storage models. The database query must explicitly join the portfolio and position data in the `WHERE` clause by matching the `position` table's foreign key, `id`, with the `portfolio` table's primary key. In the object model, this one-to-many relationship is implicitly specified by defining the `positions` map as a field of the `portfolio` class.

This difference is reflected again when you query the full list of positions market values.

Database Table Query 2	Cache Query 2

`SELECT mktValue`
`      FROM position`
	`IMPORT cacheRunner.Position;`
	`SELECT DISTINCT posnVal.mktValue`
	`   FROM /portfolios,`
	`      positions.values posnVal TYPE Position`

The `position` table is independent of the `portfolio` table, so your database query runs through the single table. The cached position data, however, is only accessible via the `portfolio` objects. The cache query aggregates data from each `portfolio` object's individual `positions` map to create the result set.

For the cache query engine, the `positions` *data only becomes visible when the first expression in the* `FROM` *clause,* `/portfolios`*, is evaluated.*

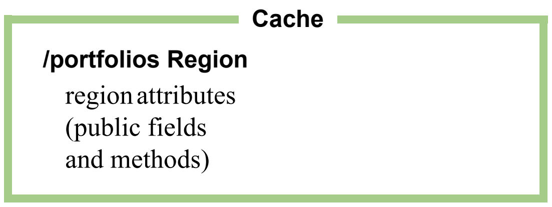
The next section explores the drill-down nature of cache querying in more detail.

Data Accessibility in a GemFire Cache Query

Before a query is run, the only cached data that is directly accessible, or in scope, are regions and region attributes. When a region or one of its attributes is referenced, the data returned by the reference comes into scope and can itself be referenced. For example, a region's keys and entries can be brought into scope by referencing `/portfolios.keySet` and `/portfolios.entrySet` in the `FROM` clause. These evaluate, respectively, to the collection of region keys and the collection of region `Entry` objects. The expression, `/portfolios`, evaluates by default to `/portfolios.values`, the collection of region entry values. This is used in the example query below.

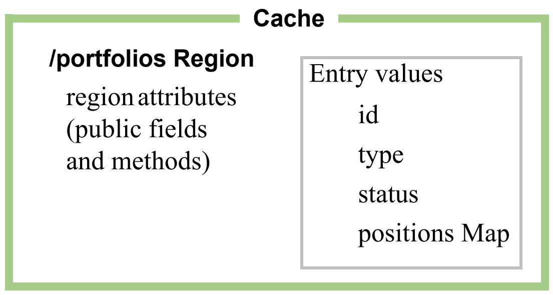
This discussion shows the `/portfolios` data that is accessible as each part of a cache query is evaluated (the query used is the first one from the prior examples). For each figure, the part of the query that has been evaluated is in **bold**. The first figure shows the data that is in scope before the query is run. Note that the region fields and methods are visible.

Figure 9.3 Data in Scope Before the Query



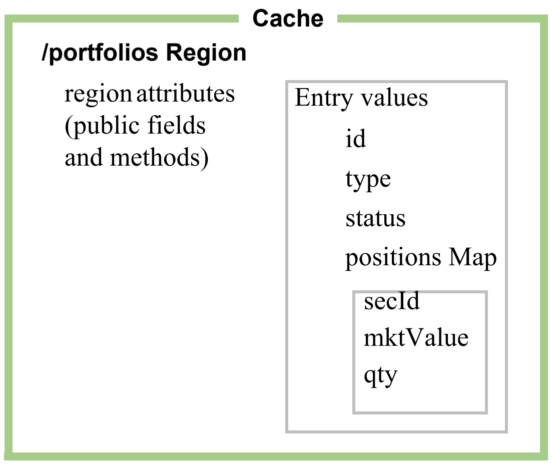
```
IMPORT cacheRunner.Position;
SELECT DISTINCT posnVal.mktValue
   FROM /portfolios,
   positions.values posnVal TYPE Position
   WHERE status='active'
```

Figure 9.4 Data in Scope After First FROM Expression is Evaluated



```
IMPORT cacheRunner.Position;
SELECT DISTINCT posnVal.mktValue
   FROM /portfolios,
   positions.values posnVal TYPE Position
   WHERE status='active'
```

Figure 9.5 Data in Scope After Second FROM Expression is Evaluated



```
IMPORT cacheRunner.Position;
SELECT DISTINCT posnVal.mktValue
   FROM /portfolios,
   positions.values posnVal TYPE Position
   WHERE status='active'
```

In cache querying, the `FROM` expression brings new data into the query scope. These concepts are discussed further in *Attribute Visibility* on page 191.

9.3 The Query Language

The query language supported in the GemFire Enterprise native client is essentially a subset of OQL (ODMG 3.0 Object Data Management Group, www.odmg.org), which is based on SQL-92. The query language provides the basic set of clauses, expressions, and operators used to retrieve elements from objects stored in cache regions. This discussion describes the operators, keywords, and operations that the query language supports.

Query language keywords such as `SELECT`*,* `DATE`*, and* `NULL`*, are case-insensitive. Identifiers in query strings, such as attribute names, method names, and path expressions, are case-sensitive. The attribute* `empId` *will only get mapped to the field* `empId` *or the methods* `getEmpId` *or* `empId`*. It will not get mapped to the method* `EMPID`*. Query language keywords are capitalized in the documentation for easy identification.*

In this section:

Accessing Cached Data

Accessing your cached data through the querying service is similar to how you would access database contents through SQL queries, but with differences. How you specify your regions and region contents is particular to the native client. The query language supports drilling down into nested object structures. Regions can contain nested data collections that are unavailable until referenced in the `FROM` clause. For a general discussion of this, see *Querying Object Data* on page 186. This discussion describes how to navigate to your cached data through the native client query service.

Querying and indexing only operate on remote cache server contents.

Basic Region Access

In the context of a query, the name of a region is specified by its full path starting with a forward slash (`/`).

Region names are restricted to alphanumeric and underscore characters only.

Object attributes

You can access the `Region` object's public fields and methods from a region path, referred to as the region's attributes. Using this method, `/portfolios.name` returns "`portfolios`" and `/portfolios.name.length` returns `10`. An attribute is mapped to a Java class member in three possible ways with the following priority until a match is found. If the attribute is named `x`, then:

```
public method getX()
public method x()
public field x
```

The term attribute *in this context is not the same as a region attribute.*

Region data

You can also access entry keys and entry data through the region:

- `/portfolios.keySet` returns the `Set` of entry keys in the region
- `/portfolios.entrySet` returns the `Set` of `Region.Entry` objects
- `/portfolios.values` returns the `Collection` of entry values
- `/portfolios` returns the `Collection` of entry values

  These collections are immutable. Invoking modifier methods on them, such as `add` *and* `remove`*, result in an* `UnsupportedOperationException`.

For the last two bullets, the `FROM` clause `/portfolios.values` and `/portfolios` return the same thing.

Drilling Down for Modifying Query Scope

The query engine resolves names and path expressions according to the name space that is currently in scope in the query. This is not the region `scope` attribute, but the scope of the query statement. The initial name space for any query is composed of the region paths of the cache on the cache server and the attributes of those paths. New name spaces are brought into scope based on the `FROM` clause in the `SELECT` statement. For example, in this query the `FROM` expression evaluates to the collection of entry values in `/portfolios`. This is added to the initial scope of the query and `status` is resolved within the new scope.

```
SELECT DISTINCT *
   FROM /portfolios
   WHERE status = 'active'
```

Each `FROM` clause expression must resolve to a collection of objects available for iteration in the query expressions that follow. In the example above, `/portfolios` resolves to the `Collection` of entry values in the region. The entry value collection is iterated by the `WHERE` clause, comparing the `status` field to the string `active`. When a match is found, the value object is added to the return set.

In the following query, the collection specified in the first `FROM` clause expression is used by the second `FROM` clause expression and by the projections of the `SELECT` statement.

```
IMPORT cacheRunner.Position;
SELECT DISTINCT "type"
   FROM /portfolios, positions.values posnVal TYPE Position
   WHERE posnVal.qty > 1000.00
```

You cannot change the order of the expressions in this `FROM` *clause. The second expression depends on the scope created by the first expression.*

Attribute Visibility

Within the current query scope, you can access any available object or object attribute. In querying, an object's attribute is any identifier that can be mapped to a public field or method in the object. In the FROM specification, any object that is in scope is valid, so at the beginning of a query all cached regions and their attributes on the cache server are in scope.

This query is valid because name resolves to the Region method getName:

```
/portfolios.name
```

This query is valid because toArray resolves to the Collection method with the same name:

```
SELECT DISTINCT * FROM /portfolios.toArray
```

You cannot, however, refer to the attribute of a collection object in the region path expression where the collection itself is specified. The following statement is invalid because neither Collection nor Region contain an attribute named positions. The entry values collection (specified by /portfolios) that does contain an attribute named positions is not yet part of the query name space.

```
/* INCORRECT: positions is not an attribute of Region or of Collection */
    SELECT DISTINCT * FROM /portfolios.positions
```

This following SELECT statement is valid because positions is an element of the entry value collection that is specified by /portfolios. The entry value collection is in scope as soon as the specification in the FROM expression is complete (before WHERE or SELECT are evaluated).

```
SELECT DISTINCT positions FROM /portfolios
```

You can also refer to positions inside the FROM clause after the /portfolios entry value collection is created. In this example, positions is an element of the /portfolios entry value collection and values is an attribute of positions:

```
IMPORT javaobject.Position;
SELECT DISTINCT posnVal
       FROM /portfolios, positions.values posnVal TYPE Position
       WHERE posnVal.mktValue >= 25.00
```

After the comma in the FROM clause, /portfolios is in scope, so its value collection can be iterated. In this case, this is done with the second FROM clause specification, positions.values. This is discussed further in *The SELECT Statement* on page 194.

Nested Query Scopes

Scopes can be nested using nested `SELECT` statements, and names in an inner scope hide identical names in an outer scope. In the query below, the inner `SELECT` creates a new scope, the `positions` of the current portfolio, inside the outer `SELECT`'s scope, `/portfolios`. This inner scope (the collection of entry values from the `/portfolios` region) is first searched for the `secId` element. The outer scope is searched only if the `secId` element is not found in the inner scope.

```
IMPORT javaobject.Position;
SELECT DISTINCT * FROM /portfolios
   WHERE NOT
   (SELECT DISTINCT * FROM positions.values TYPE Position
      WHERE secId='YYY').isEmpty
```

This statement shows the outer scope in bold. The outer scope has all the attributes of a `Portfolio` in it.

```
IMPORT javaobject.Position;
SELECT DISTINCT * FROM /portfolios
   WHERE NOT
   (SELECT DISTINCT * FROM positions.values TYPE Position
      WHERE secId='YYY').isEmpty
```

Now the statement with the inner scope is shown in bold. The inner scope has all the attributes of a `Portfolio` in it (inherited from the outer scope), and all the attributes of a `Position` as well.

```
IMPORT javaobject.Position;
SELECT DISTINCT * FROM /portfolios
   WHERE NOT
   (SELECT DISTINCT * FROM positions.values TYPE Position
      WHERE secId='YYY').isEmpty
```

When Names Cannot Be Resolved

When a query is executed and a name or path expression resolves to more than one region name in the scope, or if it cannot be resolved at all, then the client receives a `QueryException`. The `QueryException` contains the message that is generated for the exception that occurs on the server.

What is a Query String?

A query string for the GemFire Enterprise native client is a subset of OQL (ODMG 3.0 Object Data Management Group). OQL is a SQL-like language with extended functionality for querying complex objects, object attributes and methods. When you use one of the GemFire methods to create queries, you pass in a query string as an argument. To build a query string, you combine supported keywords, expressions, and operators to create an expression that returns the information that you require.

A query string follows the rules specified by the OQL query language grammar. A query can contain:

- path expressions
- attribute names
- method invocations
- operators
- literals
- the functions `IS_DEFINED`, `IS_UNDEFINED`, `ELEMENT`, `NVL` and `TO_DATE`
- `SELECT` statements

None of these elements is required in a query string. For example, a `SELECT` statement is generally thought of as the starting point for a query, but it is not essential. In the example region, `/portfolios`, the following expressions are valid queries if the region shortcut query methods are used. See *Region Shortcut Query Methods* on page 211 for more information.

`/portfolios.size > 100`	returns `true` if the region `root/portfolios` has more than 100 values in it
`/portfolios[98].id = 12`	returns `true` if the portfolio with key `98` in region `/portfolios` has id `12`, `false` if it doesn't, and `UNDEFINED` if the portfolio for that key doesn't exist

Query Strings in the Native Client

To use a query string in a native client, specify the string as a parameter in a `QueryService::newQuery` method, then execute the query using `Query::execute`, passing in the required parameters.

Alternatively, if an expression evaluates to a boolean value then you can specify it using the region shortcut methods `Region::existsValue`, `Region::selectValue`, and `Region::query`. These shortcut methods evaluate whether given expressions return any entries and return a single value entry, respectively. See *Region Shortcut Query Methods* on page 211 for more information about these shortcut methods.

If your query requires any `IMPORT` *statements, these must be included before the* `SELECT` *statement in the query string that is passed to the query engine. It should be a fully qualified package name relative to the cache server. The Java class definition must exist and have the exact footprint as the native client C++ class.*

The following sections discuss query expression elements in greater detail.

The SELECT Statement

The basic format of a `SELECT` statement is

```
SELECT DISTINCT projectionList FROM collection1, [collection2, ...]
[WHERE clause]
```

The `DISTINCT` *keyword is required.*

In the `SELECT` statement, the `FROM` clause data is input to the `WHERE` clause, and the `WHERE` clause filters the data before passing it to the `SELECT` projection list. The `SELECT` statement output is the output from the projection list operations. This is similar to SQL for database access, where the `FROM` clause specifies the data tables, the `WHERE` clause specifies the rows, and the `SELECT` clause specifies the columns.

These are the semantics of the query `SELECT` statement:

- The `FROM` clause specifies the cached data to be accessed. Each expression in this clause must evaluate to a collection, which is a group of distinct objects of homogeneous type. There must be at least one collection specified. The collections specified in the `FROM` clause are iterated over by the rest of the query, including the rest of the `FROM` clause.
- The `WHERE` clause specifies which of the elements in the `FROM` clause collections are to be passed on to the `SELECT` clause, while the rest are dropped. The `WHERE` clause specifications are called "`WHERE` search criteria." The `WHERE` clause is optional. Without it, all elements of the collections are added to the interim result.
- The projection list in the `SELECT` clause specifies the attributes to retrieve from the result elements, and may also perform transformations. The `SELECT` clause projection list can be one or more expressions, or it can be `*` to put all elements in the return set without modification.

In the following query, the `FROM` clause of this query accesses the collection of all entry values in the `/portfolios` region. This is passed to the `WHERE` clause, which searches for entry values whose `status` field is `active`. For each of the values found by the `WHERE` clause, the `SELECT` projection retrieves the value of the `type` attribute into the result set.

```
SELECT DISTINCT "type" FROM /portfolios WHERE status = 'active'
```

Query Language Reserved Words

In the query above, the `type` attribute is in double quotes to distinguish it from the reserved words of the query language. To access any method, attribute, or named object that has the same name as a query language reserved word, enclose the name within double quotation marks. Here is another example:

```
SELECT DISTINCT * FROM /region1 WHERE emps."select"() < 100000
```

A complete list of the query language reserved words is provided in Appendix C, *Query Language Grammar and Reserved Words*, on page 297.

SELECT Statement Result

The result of a `SELECT` statement is either `UNDEFINED` or is a collection that implements the `SelectResults` interface (for details, see the online native client API documentation provided in the `docs` directory). Since a `SELECT` statement returns a result, it can be composed with other expressions like the following example:

```
(SELECT DISTINCT * FROM /portfolios WHERE status = 'active').iterator
```

Query results are discussed further in *SELECT Statement Query Results* on page 200.

The rest of this section explores the `SELECT` statement elements in greater detail. The subsections are:

The FROM Clause

The `FROM` clause establishes collections of objects that are iterated over by the remainder of the query. The attributes of the objects in these collections are added to the name space scope for the remainder of the `FROM` clause as well as for the `WHERE` clause and the `SELECT` projection list. For a discussion of the effects of the `FROM` expressions on query scope, see *Drilling Down for Modifying Query Scope* on page 190.

Each `FROM` clause expression must evaluate to a collection. The expression `/portfolios.keySet` is valid because it evaluates to a `Collection`, but `/portfolios.name`, which evaluates to a `String`, causes an exception to be thrown.

> *Like the SQL query, which iterates over the tables named in its* `FROM` *clause, the OQL query iterates over the* `Collections` *established in its* `FROM` *clause.*

In the following query, `positions.values` evaluates to a Collection because `positions` is a `Map`, and the method `values` on `Map` returns a `Collection`.

```
IMPORT javaobject.Position;
SELECT DISTINCT "type"
   FROM /portfolios, positions.values posnVal TYPE Position
   WHERE posnVal.qty > 1000.00
```

> *Every expression in the* `FROM` *clause must evaluate to a* `Collection`*. For a* `Map`*, the* `values` *method returns a* `Collection`*.*

If `positions` were a `List` instead of a `Map`, this query could be used to retrieve the data:

```
IMPORT javaobject.Position;
SELECT DISTINCT "type"
FROM /portfolios, positions posnVal TYPE Position
WHERE posnVal.qty >= 1000.00
```

> *A* `List` *is a* `Collection`*, so it can be accessed directly or through its* `toArray` *method.*

> *For each object type accessed in your* `FROM` *clause, use the method that returns a* `Collection` *for that object.*

Each expression in the `FROM` clause can be any expression that evaluates to a `Collection`. An expression in the `FROM` clause is typically a path expression that resolves to a region in the cache so that the values in the region become the collection of objects to filter.

For example, this is a simple `SELECT` statement that evaluates to a set of all the entry value objects of the region `/portfolios` with active status. The collection of entry values provided by the `FROM` clause is traversed by the `WHERE` clause, which accesses each element’s `status` attribute for comparison.

```
SELECT DISTINCT * FROM /portfolios WHERE status = 'active'
```

If the `FROM` clause has just one expression in it, the result of the clause is the single collection that the expression evaluates to. If the clause has more than one expression in it, the result is a collection of structs that contain a member for each of those collection expressions. For example, if the `FROM` clause contains three expressions that evaluate to collections `C1`, `C2`, and `C3`, then the `FROM` clause generates a set of `struct(x1, x2, x3)` where `x1`, `x2`, and `x3` represent nested iterations over the collections specified. If the collections are independent of each other, this struct represents their cartesian product.

In this query, the `FROM` clause produces a struct of portfolio and position pairs to be iterated. Each element in the struct contains the portfolio and one of its contained positions.

```
IMPORT javaobject.Position;
SELECT DISTINCT "type" FROM /portfolios, positions TYPE Position
   WHERE qty > 1000.00
```

Iterator variables

For each collection expressed in the `FROM` clause, you can associate an explicit variable. The variable is added to the current scope and becomes the iterator variable bound to the elements of the collection as they are iterated over. In this example, `pflo` and `posnVal` are both explicit iterator variables.

Example 9.8 Query Using Explicit Iterator Variables

```
IMPORT javaobject.Position;
SELECT DISTINCT pflo."type", posnVal.qty
   FROM /portfolios pflo, positions.values posnVal TYPE Position
   WHERE pflo.status = 'active' and posnVal.mktValue > 25.00
```

Importing object classes

To facilitate the specification of type in variable type declarations and in typecasting expressions, a query string can have `IMPORT` statements preceding the declarations. By using `IMPORT` in the query string, the client can tell the cache server about the class definition of the serialized object that is present in the cache server region.

The only place you can have a package name in a query is in an import statement. These are valid:

```
IMPORT com.myFolder.Portfolio;

IMPORT com.myFolder.Portfolio AS MyPortfolio;
```

The first form of the import statement allows `Portfolio` to be used as the name of the class, `com.myFolder.Portfolio`. The second form provides an alternative class name, `MyPortfolio`, to be used. This is useful when a class name is not unique across packages and classes in a single query. The following example uses imported classes:

Example 9.9 Using Imported Classes

```
IMPORT com.commonFolder.Portfolio;
IMPORT com.myFolder.Portfolio AS MyPortfolio;
SELECT DISTINCT mpflo.status
FROM /portfolios pflo TYPE Portfolio,
   /myPortfolios mpflo TYPE MyPortfolio,
WHERE pflo.status = 'active' and mpflo.id = pflo.id
```

This entire query string must be passed to the query engine, including the `IMPORT` statements.

Common type names do not require an `IMPORT` statement. The following table lists the types that are defined by the system and the Java types they represent.

Table 9.3 Predefined Class Types

Type	Java Representation	Native Client Representation
`short`	`short`	`CacheableInt16`
`long`	`long`	`CacheableInt64`
`int`	`int`	`CacheableInt32`
`float`	`float`	`CacheableFloat`
`double`	`double`	`CacheableDouble`
`char`	`char`	`CacheableWideChar`
`string`	`java.lang.String`	`CacheableString`
`boolean`	`boolean`	`CacheableBoolean`
`byte` or `octet`	`byte`	`CacheableByte`
`date`	`java.sql.Date`	`CacheableDate`
`time`	`java.sql.Time`	Unsupported
`timestamp`	`java.sql.Timestamp`	Unsupported
`set<`*`type`*`>`	`java.util.Set`	`CacheableHashSet`
`list<`*`type`*`>`	`java.util.List`	`CacheableVector`
`array<`*`type`*`>`	`java.lang.Object[]`	`CacheableArray`
`map<`*`type`*`,`*`type`*`>` or `dictionary<`*`type`*`,`*`type`*`>`	`java.lang.Map`	`CacheableHashMap`

The type specification can be an imported type or any of these predefined types.

Specifying the object types of FROM clause collections

To resolve implicit attribute names, the query engine must be able to associate each attribute or method name to a single iterator expression in the `FROM` clause. Depending on the complexity of the query, the engine may be able to discover the proper associations on its own, but providing the specifications described here increases the chances for success.

The server region being queried should contain only homogeneous objects of the same type. See Setting Object Type Constraints on page 181 *for more information.*

The object type information must be available when the query is created. To provide the appropriate information to the query engine, specify the type for each of your `FROM` clause collection objects by importing the object's class before running the query and typing the object inside the query. For the example region, this query is valid (all of the examples in this chapter assume that this `IMPORT` statement is provided):

Example 9.10 Query Using IMPORT and TYPE for Object Typing

```
IMPORT javaobject.Position;
SELECT DISTINCT mktValue
   FROM /portfolios, positions.values TYPE Position
   WHERE mktValue > 25.00
```

This entire query string must be passed to the query engine, including the `IMPORT` statement. Import the object's class before running the query and typecast the object inside the query. For the example region, both of these queries are valid:

Example 9.11 Query Using IMPORT and Typecasting for Object Typing

```
IMPORT javaobject.Position;
SELECT DISTINCT value.mktValue
   FROM /portfolios, (map<string,Position>)positions
   WHERE value.mktValue > 25.00

IMPORT cacheRunner.Position;
SELECT DISTINCT mktValue
   FROM /portfolios, (collection<Position>)positions.values
   WHERE mktValue > 25.00
```

This entire query string must be passed to the query engine, including the `IMPORT` statement. Use named iterators in the `FROM` clause and explicitly prefix the path expression with iterator names.

Example 9.12 Query Using Named Iterators for Object Typing

```
SELECT DISTINCT posnVal
FROM /portfolios pflo, pflo.positions.values posnVal
WHERE posnVal.mktValue >= 25.00
```

The `IMPORT` statements in these examples assume that `$GEMFIRE/examples/dist/classes` is in the `CLASSPATH`. This is required so the cache server can process `IMPORT` statements. The class's package name cannot be used in the `FROM` clause. The package name must be specified in an `IMPORT` statement.

There is one exception to these typing guidelines. If one `FROM` expression lacks explicit typing, the query engine associates all unresolved attributes with that expression and creates the query. An exception is thrown if any of these attributes are not found at execution time.

The WHERE Clause

The optional `WHERE` clause defines the search criteria for the selection. The `WHERE` clause refines the search to filter the set of elements specified by the `FROM` clause. Without a `WHERE` clause, the `SELECT` projection list receives the entire collection or set of collections as specified in the `FROM` clause.

The query processor searches the collection for elements that match the conditions specified in the `WHERE` clause conditions. If there is an index on an expression matched by the WHERE clause, then the query processor may use the index to optimize the search and avoid iterating over the entire collection. For more information on indexes, see the *Querying and Indexing* chapter in the *GemFire Enterprise Developer's Guide*.

A `WHERE` clause expression is a boolean condition that is evaluated for each element in the collection. If the expression evaluates to `true` for an element, the query processor passes that element on to the `SELECT` projection list. This example uses the `WHERE` clause to return the portfolio objects in the region that have a type `xyz`.

```
SELECT DISTINCT * FROM /portfolios WHERE "type" = 'xyz'
```

The next query returns the set of all portfolios with a type of `xyz` and active status.

```
SELECT DISTINCT * FROM /portfolios WHERE "type" = 'xyz' AND status =
'active'
```

Joins

If collections in the `FROM` clause are not related to each other, the `WHERE` clause can be used to join them. The statement below returns all the persons from the `/Persons` region with the same name as a flower in the `/Flowers` region.

```
SELECT DISTINCT p FROM /Persons p, /Flowers f WHERE p.name = f.name
```

Indexes are supported for region joins. To create indexes for region joins, you create single-region indexes for both sides of the join condition. These are used during query execution for the join condition.

The SELECT Projection List

The projections in the `SELECT` project list are used to transform the results of the `WHERE` search operation. The projection list is either specified as * or as a comma delimited list of expressions. For *, the interim results of the `WHERE` clause are returned from the query. Otherwise, the set of objects in the interim results are iterated and the projections applied to each of the objects. During the application of the projection list, the attributes of the objects being traversed are in scope for name resolution.

You can also specify retrieval of the entry keys in your projection list. This allows you to access the associated cached entries for modification and other purposes. The following example shows how the `Region` entry key can be obtained by using the region entries in the `FROM` clause and using appropriate projections. This query runs on the `/portfolios` region, returning a set of `struct<key:string, id:string, secId:string>` where `key` is the key of the region entry, `id` is an entry ID, and `secId` is a secId of a `positions` map for the entry:

```
SELECT DISTINCT key, entry.value.id, posnVal.secId
   FROM /portfolios.entrySet entry, entry.value.positions.values posnVal
   WHERE entry.value."type" = 'xyz' AND posnVal.secId = 'XXX'
```

You can assign arbitrary names to the return values by using the `fieldname:expression` syntax in the projection list. This modification to the query results in a set of `struct<newKey:string, newId:string, newSecId:string>`:

```
SELECT DISTINCT newKey: entry.key, newId: entry.value.id,
      newSecId: posnVal.secId
   FROM /portfolios.entrySet entry,
      entry.value.positions.values posnVal
   WHERE entry.value."type" = 'xyz' AND posnVal.secId = 'XXX'
```

SELECT Statement Query Results

The result of a `SELECT` statement is a collection that implements the `SelectResults` interface (see the online API documentation for `Query`) or it is `UNDEFINED`. The `SelectResults` returned from the `SELECT` statement is either a collection of objects or a `Struct` collection containing the objects.

A `Collection` of objects is returned in two cases:

- When only one expression is specified by the projection list and that expression is not explicitly specified using the `fieldname:expression` syntax
- When the `SELECT` list is `*` and a single collection is specified in the `FROM` clause

In all other cases, a `Struct` is generated with a field for each expression in the `SELECT` projection list. This table describes the contents of the `SelectResults` based on the `SELECT` projection list and the expressions in the `FROM` clause.

Table 9.4 Matrix of SelectResults Contents Based on SELECT and FROM Clause Specifications

SELECT / FROM	*	Single Expression	Multiple Expressions
single expression	Objects	Objects. (`Struct` if the projection specifies a field name.)	`Struct`
multiple expressions	`Struct`	Objects. (`Struct` if the projection specifies a field name.)	`Struct`

When a `Struct` is returned, the name of each field in the `Struct` is determined as follows:

- If a field is specified explicitly using the `fieldname:expression` syntax, the `fieldname` is used.
- If the `SELECT` projection list is * and an explicit iterator expression is used in the `FROM` clause, the iterator variable name is used as the field name.
- If the field is associated with a region or attribute path expression, the last attribute name in the expression is used.

If names can not be decided based on these rules, arbitrary unique names are generated by the query processor.

These examples show how the projections and `FROM` clause expressions are applied:

`SELECT <*> FROM <single expression>`	`SELECT DISTINCT *`
	`    FROM /portfolios`
	`    WHERE status ='active'`
		Returns the `Collection` of active portfolios objects.

`SELECT <single expression> FROM <multiple expression>` (without `fieldName` mentioned)	`IMPORT javaobject.Position;`
	`SELECT DISTINCT secId`
	`    FROM /portfolios, positions.values TYPE Position`
	`    WHERE status ='active'`
		Returns the `Collection` of `secIds` (`CacheableString` objects) from the positions of active portfolios.

`SELECT <single expression> FROM <multiple expression>` (with `fieldName` mentioned)	`IMPORT javaobject.Position;`
	`SELECT DISTINCT secIdFieldName:secId`
	`    FROM /portfolios, positions.values TYPE Position`
	`    WHERE status ='active'`
		Returns `struct<secIdField: CacheableString>` for the active portfolios. (Compare to the results for the prior query.)

`SELECT <multiple expression> FROM <single expression>`	`SELECT DISTINCT "type", positions`
	`    FROM /portfolios`
	`    WHERE status ='active'`
		Returns `struct<type: CacheableString, positions: CacheableHashMap Position>` for the active portfolios. The second field of the `struct` is a `CacheableHashMap` object, which contains `secIds` as keys and `Position` objects as values.

`SELECT <*> FROM <multiple expression>`	`IMPORT javaobject.Position;`
	`SELECT DISTINCT *`
	`FROM /portfolios, positions.values TYPE Position`
	`WHERE status = 'active'`
		Returns a `Collection` of `struct<portfolios: Portfolio, values: Position>` for the active portfolios.

`SELECT <multiple expression> FROM <multiple expression>`	`IMPORT javaobject.Position;`
	`SELECT DISTINCT pflo, posn`
	`    FROM /portfolios pflo, positions posn TYPE Position`
	`    WHERE pflo.status = 'active'`
		Returns a `Collection` of `struct<pflo: Portfolio, posn: Position>` for the active portfolios.

Additional Query Language Elements

This section discusses various aspects and tools of the native client query engine. The sections are:

Method Invocation

The query language supports method invocation inside query expressions. The query processor maps attributes in query strings using the attribute rules described in *Object attributes* on page 189.

> *Methods declared to return* `void` *evaluate to* `null` *when invoked through the query processor.*

Invoking methods without parameters

If you know that the attribute name maps to a public method that takes no parameters, you can simply include the method name in the query string as an attribute. For example, `emps.isEmpty` is equivalent to `emps.isEmpty()`. In the following example, the query invokes `isEmpty` on `positions`, and returns the set of all portfolios with no positions.

```
SELECT DISTINCT * FROM /portfolios WHERE positions.isEmpty
```

Invoking methods with parameters

The native client also supports the invocation of public methods with parameters. To invoke methods with parameters, include the method name in the query string as an attribute and provide the method arguments between parentheses. Only constants as part of the query strings are possible.

Operators

GemFire supports the following operator types in expressions:

- Comparison operators (page 203)
- Logical operators (page 203)
- Unary operators (page 204)
- Map and index operators (page 204)
- Dot and forward slash operators (page 204)

Comparison operators

Comparison operators compare two values and return the results as either `true` or `false`. The native client query language supports the following comparison operators:

=	equal to	<	less than
<>	not equal to	<=	less than or equal to
!=	not equal to	>	greater than
		>=	greater than or equal to

The equal (=) and not equal (<>, `!=`) operators have lower precedence than the other comparison operators.

The equal (=) and not equal (<>, `!=`) operators can be used with `NULL`. To perform equality or inequality comparisons with `UNDEFINED`, use the `IS_DEFINED` and `IS_UNDEFINED` operators. For more information, refer to *Rules for UNDEFINED* on page 205.

The following example selects all the portfolios with more than one position:

```
SELECT DISTINCT * FROM /portfolios WHERE positions.size >= 2
```

This example selects all the portfolios whose status is active:

```
SELECT DISTINCT * FROM /portfolios WHERE status = 'active'
```

This example selects all the portfolios whose id is not ‘`XYZ-1`’:

```
SELECT DISTINCT * FROM /portfolios WHERE id <> 'XYZ-1'
```

Logical operators

The operators `AND` and `OR` allow you to create more complex expressions by combining expressions to produce a boolean result (`TRUE` or `FALSE`). When you combine two conditional expressions using the `AND` operator, both conditions must evaluate to true for the entire expression to be true. When you combine two conditional expressions using the `OR` operator, the expression evaluates to true if either one or both of the conditions are true. You can create complex expressions by combining multiple simple conditional expressions with `AND` and `OR` operators. When expressions are combined using multiple `AND` and `OR` operators, the `AND` operator has higher precedence than the `OR` operator.

The following query string selects only those portfolios with a type of ‘`xyz`’ that have active status. If either condition is false for an element in the queried collection portfolios, that element is not included in the result.

```
SELECT DISTINCT * FROM /portfolios
   WHERE "type" = 'xyz' AND status = 'active'
```

The following query string selects only those portfolios whose ID is `XYZ-1` or whose ID is `ABC-1`. If either one of these conditions is true for an element of portfolios, that element is included in the result.

```
SELECT DISTINCT * FROM /portfolios
   WHERE id = 'XYZ-1' OR id = 'ABC-1'
```

In the following example, the query selects only those portfolios with positions that have a market value over `30.00` and that have an ID of `XYZ-1` or `ABC-1`.

```
IMPORT cacheRunner.Position;
SELECT DISTINCT * FROM /portfolios, positions.values TYPE Position
   WHERE mktValue > 30.00
   AND (id = 'XYZ-1' OR id = 'ABC-1')
```

Unary operators

Unary operators operate on a single value or expression, and have lower precedence than comparison operators in expressions. The native client supports the unary operator `NOT`. `NOT` is the negation operator, which changes the value of the operand to its opposite. That is, if the expression evaluates to `TRUE`, `NOT` changes this value to `FALSE`. The operand must be a boolean. The following query returns the set of portfolios that have positions.

```
SELECT DISTINCT * FROM /portfolios WHERE NOT positions.isEmpty
```

Map and index operators

Map and index operators access elements in key/value collections (such as maps and regions) and ordered collections (such as arrays, lists, and `String`s). The operator is represented by a set of square brackets "`[ ]`" immediately following the name of the collection. The mapping or indexing specification is provided inside these brackets. Map operators are derived from `java.utils.map` interface methods.

- Array, list, and `String` elements are accessed using an index value. Indexing starts from zero, with `0` representing the first element, `1` the second element, and so on. If `myList` is an array, list, or `String` and `index` is an expression that evaluates to a non-negative integer, then `myList[index]` represents the (`index+1`)th element of `myList`. The elements of a `String` are the list of characters that make up the string.
- Map and region values are accessed by key using the same syntax. The key can be any `Object`. In the case of a `Region`, the map operator performs a non-distributed `get` that does not cause a `netSearch` if the value is not present in the local cache. Thus, `myRegion[keyExpression]` is the equivalent of `myRegion.getEntry(keyExpression).getValue`.

Dot and forward slash operators

The dot operator separates attribute names in a path expression, and specifies the navigation through object attributes. An alternative equivalent to the dot is the right arrow, "`->`".

Rules for UNDEFINED

The special value `UNDEFINED` behaves according to the following rules:

- Accessing an attribute of a `NULL` value results in `UNDEFINED`.
- `IS_UNDEFINED(UNDEFINED)` returns `TRUE`.
- `IS_DEFINED(UNDEFINED)` returns `FALSE`.
- When the predicate defined by a `WHERE` clause returns `UNDEFINED`, this is handled as if the predicate returns `FALSE`.
- `UNDEFINED` is a valid value of explicit construction expressions (see *Construction Expressions* on page 206) and can be in the results of a `SELECT` statement if the value of a projection expression is `UNDEFINED`.
- For AND and OR expressions:

Expression	Return Value
`UNDEFINED AND FALSE`	`FALSE`
`UNDEFINED AND TRUE`	`UNDEFINED`
`UNDEFINED AND UNDEFINED`	`UNDEFINED`
`UNDEFINED OR FALSE`	`UNDEFINED`
`UNDEFINED OR TRUE`	`TRUE`
`UNDEFINED OR UNDEFINED`	`UNDEFINED`

Any other operation with any `UNDEFINED` operands results in `UNDEFINED`. This includes the dot operator, method invocations with `UNDEFINED` arguments, and comparison operations. For example, the operation `X=UNDEFINED` evaluates to `UNDEFINED`, but `IS_UNDEFINED(UNDEFINED)` evaluates to `TRUE`.

Functions

The query language supports these functions:

`ELEMENT(query)`—Extracts a single element from a collection or array. This function throws a `FunctionDomainException` if the argument is not a collection or array with exactly one element.

`IS_DEFINED(query)`—Returns `TRUE` if the expression does not evaluate to `UNDEFINED`.

`IS_UNDEFINED(query)`—Returns `TRUE` if the expression evaluates to `UNDEFINED`.

`NVL(expr1, expr2)`—Returns `expr2` if `expr1` is `null`. The expressions can be bind arguments, path expressions, or literals.

`TO_DATE (date_str, format_str)`—Returns a Java `Date` class object. The arguments must be `String`s with `date_str` representing the date and `format_str` representing the format used by `date_str`.

For information on defined and undefined values, see *Rules for UNDEFINED* on page 205.

The following example returns a boolean indicating whether the single portfolio with ID `XYZ-1` is active.

```
ELEMENT(SELECT DISTINCT * FROM /portfolios
      WHERE id = 'XYZ-1').status = 'active'
```

The next example returns all portfolios with an undefined value for status. For example, if status is `null`, it is undefined.

```
SELECT DISTINCT * FROM /portfolios
      WHERE IS_UNDEFINED(status)
```

In most queries, undefined values are not included in the query results. The `IS_UNDEFINED` function allows undefined values to be included in a result set, so you can retrieve information about elements with undefined values (or at least identify elements with undefined values).

Construction Expressions

The construction expression implemented in the cache server is the set constructor. A set can be constructed using `SET(e1, e2, ..., en)` where `e1, e2, ..., en` are expressions. This constructor creates and returns the set containing the elements `e1, e2, ..., en`. For example, `SET(1, 2, 3)` returns a `Set` containing the three elements `1`, `2`, `3`.

The IN Expression

The `IN` expression is a boolean indicating if one expression is present inside a collection of expressions of compatible type.

If `e1` and `e2` are expressions, `e2` is a collection, and `e1` is an object or a literal whose type is a subtype or the same type as the elements of `e2`, then

```
e1 IN e2
```

is an expression of type `boolean`. The expression returns:

- `TRUE`, if `e1` is not `UNDEFINED` and belongs to collection `e2`
- `FALSE`, if `e1` is not `UNDEFINED` and does not belong to collection `e2`
- `UNDEFINED`, if `e1` is `UNDEFINED`

A very simple example of this is the expression, `2 IN SET(1, 2, 3)`, which returns `TRUE`.

For a more interesting example, look at the case where the collection you are querying into is defined by a sub-query. Suppose that, in addition to the `/portfolios` region, there is another region named `/company` with attributes `id`, `name`, and `address`. You can run the following query to retrieve the names and addresses of all companies for whom you have an active portfolio on file.

```
SELECT name, address FROM /company
WHERE id IN (SELECT id FROM /portfolios WHERE status = 'active')
```

The interior `SELECT` statement returns a collection of `id`s for all `/portfolios` entries whose status is active. The exterior `SELECT` iterates over `/company`, comparing each entry's `id` with this collection. For each entry, if the `IN` expression returns `TRUE`, the associated `name` and `address` are added to the outer `SELECT`'s collection.

Literals

Query language expressions can contain literals as well as operators and attribute names. These are the literal types that the native client supports.

`boolean`—A boolean value, either `TRUE` or `FALSE`

`integer` and `long`—An integer literal is of type `long` if it is suffixed with the ASCII letter `L`. Otherwise it is of type `int`.

`floating point`—A floating-point literal is of type `float` if it is suffixed with an ASCII letter `F`. Otherwise its type is `double` and it can optionally be suffixed with an ASCII letter `D`. A double or floating point literal can optionally include an exponent suffix of `E` or `e`, followed by a signed or unsigned number.

`string`—String literals are delimited by single quotation marks. Embedded single quotation marks are doubled. For example, the character string `'Hello'` evaluates to the value `Hello`, while the character string `'He said, ''Hello'''` evaluates to `He said, 'Hello'`. Embedded newlines are kept as part of the string literal.

`char`—A literal is of type `char` if it is a string literal prefixed by the keyword `CHAR`, otherwise it is of type `string`. The `CHAR` literal for the single quotation mark character is `CHAR ''''` (four single quotation marks).

`date`—A `java.sql.Date` object that uses the JDBC format prefixed with the `DATE` keyword: `DATE yyyy-mm-dd` In the `Date`, `yyyy` represents the year, `mm` represents the month, and `dd` represents the day. The year must be represented by four digits; a two-digit shorthand for the year is not allowed.

`time`—Not supported.

`timestamp`—Not supported.

`NIL`—Equivalent alternative of `NULL`.

`NULL`—The same as `null` in Java.

`UNDEFINED`—A special literal that is a valid value for any data type. An `UNDEFINED` value is the result of accessing an attribute of a null-valued attribute. Note that if you access an attribute that has an explicit value of `null`, then it is not undefined. For example if a query accesses the attribute `address.city` and `address` is `null`, then the result is undefined. If the query accesses `address`, then the result is not undefined, it is null. For more information, see *Rules for UNDEFINED* on page 205.

Comparing values with java.util.Date

You can compare the temporal literal value `DATE` with `java.util.Date` values. However, there is no literal for `java.util.Date` in the query language.

No literal exists in the case of a data string mentioned inside a query string as constants. However, doing a comparison using the native client built-in object `CacheableDate` *results in a comparison of the date portions of the objects.*

Type Conversions

Java rules used within a query string dictate that the query processor performs implicit conversions and promotions under certain cases in order to evaluate expressions that contain different types. The query processor performs binary numeric promotion, method invocation conversion, and temporal type conversion.

Binary numeric promotion

Binary numeric promotion widens all operands in a numeric expression to the widest representation used by any of the operands. In each expression, the query processor applies the following rules in order:

- If either operand is of type `double`, the other is converted to `double`.
- If either operand is of type `float`, the other is converted to `float`.
- If either operand is of type `long`, the other is converted to `long`.
- Both operands are converted to type `int`.

The query processor performs binary numeric promotion on the operands of the following operators:

- comparison operators <, <=, >, and >=
- equality operators = and <>

This is essentially the same behavior as in Java, except that chars are not considered to be numeric in the native client query language.

Method invocation conversion

Method invocation conversion in the query language follows the same rules as Java method invocation conversion, except that the query language uses runtime types instead of compile time types, and handles null arguments differently than in Java. One aspect of using runtime types is that an argument with a null value has no typing information, and so can be matched with any type parameter. When a `null` argument is used, if the query processor cannot determine the proper method to invoke based on the non-null arguments, it throws an `AmbiguousNameException`. For more information on method invocation in query strings, see *Method Invocation* on page 202.

Temporal type conversion

The temporal types that the query language supports on the cache server include the Java types `java.util.Date` and `java.sql.Date`, which are treated the same and can be freely compared and used in indexes. When compared with each other, these types are all treated as nanosecond quantities.

Comments

You can include comments in the query string. To insert a one-line comment, begin the line with two dashes `(--)`. To insert a multiple-line comment, or to embed a comment in a line of code, begin the comment with `/*` and end it with `*/`. The following query strings include comments:

```
SELECT DISTINCT * FROM /portfolios /* here is a comment */
   WHERE status = 'active'

SELECT DISTINCT * FROM /portfolios
   WHERE status = 'active'-- here is another comment
```

9.4 Indexes

Indexes are created and maintained on the cache server. An index can provide significant performance gains for query execution. A query run without the aid of an index iterates through every object in the collection on the cache server. If an index is available that matches part or all of the query specification, the query iterates only over the indexed set, and query processing time can be reduced.

When you create your indexes on the cache server, remember that indexes incur maintenance costs as they must be updated when the indexed data changes. An index that requires many updates and is not used very often may require more system resources than no index at all. Indexes also consume memory. For information on the amount of memory used for indexes, see the system configuration information in the *GemFire Enterprise System Administrator’s Guide*.

An index for remote querying can be declaratively created on the cache server using a `cache.xml` file, as shown in the next example:

Example 9.13 Creating an Index on a Cache Server Using a Server XML File

```
<region name="portfolios">
   <region-attributes . . . >
      <value-constraint>cacheRunner.Portfolio</value-constraint>
   </region-attributes>
   <index name="myFuncIndex">
      <functional from-clause="/portfolios" expression="status"/>
   </index>
   <index name="myPrimIndex">
      <primary-key field="id"/>
   </index>
   <entry> . . .
```

For detailed information about working with indexes configured on a cache server, see the *Querying and Indexing* chapter in the *GemFire Enterprise Developer’s Guide*.

9.5 Performance Considerations

Similar to the way query processors are run against relational databases, the way a query is written can greatly effect execution performance. These are some of the things to consider when optimizing your queries for performance.

- Indexes should be created for large work sets that have a large number of region entries.
- Use indexes to improve the performance of joins, sub-queries and co-related queries that are applied across multiple regions. Indexes can also be created on partial `WHERE` clauses.
- Indexes should be carefully chosen because the memory overhead for an index can hinder the data storage capacity of the server. Also, too many indexes can result in latency for puts if the indexes are maintained synchronously.
- Embed the key inside the object for easier joins, as shown in this querying example:

  ```
  select distinct p from /Portfolios p, /Portfolios2 p2 where
  p.Mkey = p2.Mkey
  ```

  To do this, set the class definition:

  ```
  Class Portfolio {
     ID,
     Mkey,
     Qty
     ...
  }
  ```

  Next, put the data:

  ```
  ...
  Main() {
  ...
  Portfolio port=new Portfolio (key, id, qty);
  regionptr->put (key, port);
  ...
  }
  ```

- Derived `selectResults` in a query should be used with caution, since execution time cannot be optimized based on dynamic results. This example uses derived `selectResults`:

  ```
  select distinct derivedProjAttrbts,
  key: p.key, id: p.value.ID  from /Portfolios.entries p,
  (select distinct x.ID, x.status, x.getType, myPos.secId from /Portfolios
  x, x.positions.values as myPos) derivedProjAttrbts where p.value.ID =
  derivedProjAttrbts.ID and derivedProjAttrbts.secId = 'IBM'")
  ```

  Simplify these types of queries by optimizing the class relations or data storage. In the previous example, performance is improved if positions are kept in a separate region with a pointer to `Portfolios`. This simpler query determines which position objects belong to a particular portfolio:

  ```
  select distinct port from /Portfolios port, /Positions pos where
  pos.portfolio() = port.ID
  ```

- Some queries may take more time to execute on the server, and the results size may be large and take more time to be transmitted back to the client. Specify an optional timeout parameter in the query methods to allow sufficient time for the operation to succeed.

9.6 The Remote Query API

The native client querying API allows you to access all the querying functionality discussed in the previous sections. This section gives a general overview of the interfaces and classes that are provided by the `Query` package API, and the shortcut methods provided in the `Region` interface. Complete, current information on the classes and interfaces listed here is available in the native client online API documentation.

Creating and Managing Queries

QueryService

This method is the entry point to the query package. To execute a query you must obtain a `QueryService` from the `cache`. It is retrieved from the `Cache` instance through `Cache::getQueryService`. If you are using the `Pool` API you must obtain the `QueryService` from the pools and not from the cache.

Query

A `Query` is obtained from a `QueryService`, which is obtained from the `Cache`. The `Query` interface provides methods for managing the compilation and execution of queries, and for retrieving an existing query string.

A `Query` object must be obtained for each new query. The following example demonstrates the method used to obtain a new instance of `Query`:

```
QueryPtr newQuery(const char * querystr);
```

Region Shortcut Query Methods

The `Region` interface has several query shortcut methods. All take a query predicate which is used in the `WHERE` clause of a standard query. See *The WHERE Clause* on page 198 for more information. Each of the following examples also set the query response timeout to 10 seconds to allow sufficient time for the operation to succeed.

- The `query` method retrieves a collection of values satisfying the query predicate. This call retrieves active portfolios, which in the sample data are the portfolios with keys 111, 222, and 333:

  ```
  SelectResultsPtr results = regionPtr->query("status  'active' ");
  ```

- The `selectValue` method retrieves one value object. In this call, you request the portfolio with ID `ABC-1`:

  ```
  SerializablePtr port = region->selectValue("ID='ABC-1'");
  ```

- The `existsValue` method returns a boolean indicating if any entry exists that satisfies the predicate. This call returns `false` because there is no entry with the indicated `type`:

  ```
  bool entryExists = region->existsValue("'type' = 'QQQ' ");
  ```

For more information about these shortcut query methods, see the `Region` class description in the native client online API documentation.

Query Result Sets

SelectResults

This method executes the query on the cache server and returns the results as either a `ResultSet` or a `StructSet`.

SelectResultsIterator

This method iterates over the items available in a `ResultSet` or `StructSet`.

ResultSet

A ResultSet is obtained after executing a Query, which is obtained from a QueryService that is obtained from a Cache class.

StructSet

This is used when a `SELECT` statement returns more than one set of results. This is accompanied by a `Struct`, which provides the `StructSet` definition and contains its field values.

9.7 Programming Examples

In this section:

Query Management

Queries are created on the cache server through `QueryService` and then managed through the resulting `Query` object. The `newQuery` method for the `Query` interface binds a query string. By invoking the `execute` method, the query is submitted to the cache server and returns `SelectResults`, which is either a `ResultSet` or a `StructSet`.

Query Code Samples Returning ResultSet

The following API examples demonstrate methods for returning `ResultSet` for both built-in and user-fined data types.

Example 9.14 Query Returning a ResultSet for a Built-In Data Type

```
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
QueryPtr query =
   qrySvcPtr->newQuery("select distinct pkid from /Portfolios");
//specify 10 seconds for the query timeout period
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR)
{
   printf( "\nNo results returned from the server");
}
//obtaining a handle to resultset
ResultSetPtr rs(dynamic_cast<ResultSet*> (results.ptr()));
if (rs == NULLPTR)
{
printf ("\nResultSet is not obtained \n"); return;
}
//iterating through the resultset using row index.
for (int32_t row=0; row < rs->size(); row++)
{
   SerializablePtr ser((*rs)[row]);
   CacheableStringPtr str(dynamic_cast<CacheableString*> (ser.ptr()));
   if (str != NULLPTR)
   {
      printf("\n string column contains - %s \n", str->asChar() );
   }
}
```

Example 9.15 Query Returning a ResultSet for a User-Defined Data Type

```
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring = "select distinct * from /Portfolios";
QueryPtr query = qrySvcPtr->newQuery(querystring);
//specify 10 seconds for the query timeout period
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR)
{
   printf( "\nNo results returned from the server");
}
//obtaining a handle to resultset
ResultSetPtr rs(dynamic_cast<ResultSet*> (results.ptr()));
if (rs == NULLPTR)
{
   printf ("\nResultSet is not obtained \n"); return;
}
//iterating through the resultset using iterators.
SelectResultsIterator iter = rs->getIterator();
while (iter.hasNext())
{
   SerializablePtr ser = iter.next();
   PortfolioPtr port(dynamic_cast<Portfolio*> (ser.ptr()));
   if (port != NULLPTR)
   {
      printf("\nPortfolio object is - %s \n", port->toString()->asChar() );
   }
} // end of rows
```

Query Code Samples Returning StructSet

The following examples return a `StructSet` for built-in and user-defined data types, `Struct` objects, and collections.

Example 9.16 Query Returning a StructSet for a Built-In Data Type

```
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring =
   "select distinct ID, pkid, status, getType from /Portfolios";
QueryPtr query = qrySvcPtr->newQuery(querystring);
//specify 10 seconds for the query timeout period
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR)
{
   printf( "\nNo results returned from the server");
}
//obtaining a handle to resultset
StructSetPtr ss(dynamic_cast<StructSet*> (results.ptr()));
if (ss == NULLPTR)
{
   printf ("\nStructSet is not obtained \n");
   return;
}
//iterating through the resultset using indexes.
for ( int32_t row=0; row < ss->size();  row++)
{
   Struct * siptr = (Struct*) dynamic_cast<Struct*> ( ((*ss)[row]).ptr() );
   if (siptr == NULL)
   {
      printf("\nstruct is empty \n");
      continue;
   }
   //iterate through fields now
   for( int32_t field=0; field < siptr->length(); field++)
   {
      SerializablePtr fieldptr((*siptr)[field]);
      if(fieldptr == NULLPTR )
      {
         printf("\nnull data received\n");
      }
      CacheableStringPtr
         str(dynamic_cast<CacheableString*>(fieldptr.ptr()));
      if (str == NULLPTR)
      {
         printf("\n field is of some other type \n");
      }
      else
      {
         printf("\n Data for %s is %s ", siptr->getFieldName(field), str-
>asChar() );
      }
   } //end of columns
} // end of rows
```

Example 9.17 Query Returning a StructSet for a User-Defined Data Type

```
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring =
   "select distinct port.ID, port from /Portfolios port";
QueryPtr query = qrySvcPtr->newQuery(querystring);
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR)
{
   printf( "\nNo results returned from the server");
}
//obtaining a handle to resultset
StructSetPtr ss(dynamic_cast<StructSet*> (results.ptr()));
if (ss == NULLPTR)
{
   printf ("\nStructSet is not obtained \n");
   return;
}
//iterating through the resultset using indexes.
for ( int32_t row=0; row < ss->size();  row++)
{
   Struct * siptr = (Struct*) dynamic_cast<Struct*> ( ((*ss)[row]).ptr() );
   if (siptr == NULL)
   {
      printf("\nstruct is empty \n");
      continue;
   }
   //iterate through fields now
   for( int32_t field=0; field < siptr->length(); field++)
   {
      SerializablePtr fieldptr((*siptr)[field]);
      if(fieldptr == NULLPTR )
      {
         printf("\nnull data received\n");
      }
      CacheableStringPtr
         str(dynamic_cast<CacheableString*>(fieldptr.ptr()));
      if (str != NULLPTR)
      {
         printf("\n Data for %s is %s ", siptr->getFieldName(field), str-
>asChar() );
      }
      else
      {
         PortfolioPtr port(dynamic_cast<Portfolio*> (fieldptr.ptr()));
         if (port == NULLPTR)
         {
            printf("\n field is of some other type \n");
         }
         else
         {
            printf("\nPortfolio data for %s is %s \n", siptr-
>getFieldName(field),port->toString()->asChar());
         }
      } //end of columns
   }//end of rows
}
```

Example 9.18 Returning Struct Objects

```
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring =
   "select distinct derivedProjAttrbts, key: p.key from "
   "/Portfolios.entries p, (select distinct x.ID, myPos.secId from "
   "/Portfolios x, x.positions.values as myPos) derivedProjAttrbts where "
   "p.value.ID = derivedProjAttrbts.ID and derivedProjAttrbts.secId =
'IBM'";
QueryPtr query = qrySvcPtr->newQuery(querystring);
//specify 10 seconds for the query timeout period
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR)
{
   printf( "\nNo results returned from the server");
}
//obtaining a handle to resultset
StructSetPtr ss(dynamic_cast<StructSet*> (results.ptr()));
if (ss == NULLPTR)
{
   printf ("\nStructSet is not obtained \n");
   return;
}
//iterating through the resultset using indexes.
for ( int32_t row=0; row < ss->size();  row++)
{
   Struct * siptr = (Struct*) dynamic_cast<Struct*> ( ((*ss)[row]).ptr() );
   if (siptr == NULL) { printf("\nstruct is empty \n"); }
   //iterate through fields now
   for( int32_t field=0; field < siptr->length(); field++) {
      SerializablePtr fieldptr((*siptr)[field]);
      if(fieldptr == NULLPTR )
      {
         printf("\nnull data received\n");
      }
      CacheableStringPtr
         str(dynamic_cast<CacheableString*>(fieldptr.ptr()));
      if (str != NULLPTR) {
         printf("\n Data for %s is %s ", siptr->getFieldName(field),
            str->asChar() );
      }
      else
      {
         StructPtr simpl(dynamic_cast<Struct*> (fieldptr.ptr()));
         if (simpl == NULLPTR)
         {
            printf("\n field is of some other type \n"); continue;
         }
         printf( "\n struct received %s \n", siptr->getFieldName(field) );
         for( int32_t inner_field=0; inner_field < simpl->length();
inner_field++ )
         {
            SerializablePtr innerfieldptr((*simpl)[inner_field]);
            if (innerfieldptr == NULLPTR)
            {
               printf("\nfield of struct is NULL\n");
            }
            CacheableStringPtr
               str(dynamic_cast<CacheableString*>(innerfieldptr.ptr()));
```

```
                    if (str != NULLPTR)
                    {
                       printf("\n Data for %s is %s ",
                          simpl->getFieldName(inner_field),str->asChar() );
                    }
                    else
                    {
                       printf("\n some other object type inside struct\n");
                    }
                 }
              }
           } //end of columns
        }//end of rows
```

Example 9.19 Returning Collections

```
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring = "select distinct ID, names from /Portfolios";
QueryPtr query = qrySvcPtr->newQuery(querystring);
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR) {
   printf( "\nNo results returned from the server");
}
//obtaining a handle to resultset
StructSetPtr ss(dynamic_cast<StructSet*> (results.ptr()));
if (ss == NULLPTR) {
   printf ("\nStructSet is not obtained \n");
   return;
}
//iterating through the resultset using indexes.
for ( int32_t row=0; row < ss->size();  row++)
{
   Struct * siptr = dynamic_cast<Struct*> ( ((*ss)[row]).ptr() );
   if (siptr == NULL)
   {
      printf("\nstruct is empty \n");
      continue;
   }
   //iterate through fields now
   for( int32_t field=0; field < siptr->length(); field++)
   {
      SerializablePtr fieldptr((*siptr)[field]);
      if(fieldptr == NULLPTR )
      {
         printf("\nnull data received\n");
      }
      CacheableStringPtr
         str(dynamic_cast<CacheableString*>(fieldptr.ptr()));
      if (str != NULLPTR)
      {
         printf("\n Data for %s is %s ", siptr->getFieldName(field),
                 str->asChar() );
      }
      else
      {
         CacheableObjectArrayPtr
                 coa(dynamic_cast<CacheableObjectArray*>(fieldptr.ptr()));
         if (coa == NULLPTR)
         {
            printf("\n field is of some other type\n"); continue;
         }
         printf( "\n objectArray received %s \n", siptr-
>getFieldName(field) );
         for(unsigned arrlen=0; arrlen < (uint32_t)coa->length(); arrlen++)
         {
            printf("\n Data for %s is %s ",siptr->getFieldName(field),
                    coa->operator[](arrlen)->toString()->asChar());
         }
      }
   } //end of columns
}//end of rows
```

Chapter 10 Continuous Querying

Continuous querying in GemFire Enterprise® native client gives C++ and C# .NET clients a way to run queries against events in the GemFire cache server region. The clients register interest in events using simple query expressions. Events are sent to client listeners that you can program to do whatever your application requires.

Continuous queries (CQs) provide these main features:

- **Standard GemFire Enterprise native client query syntax and semantics**—CQ queries are expressed in the same language used for other native client queries (see Chapter 9, *Remote Querying*, on page 179).
- **Standard GemFire Enterprise events-based management of CQ events**—The event handling used to process CQ events is based on the standard GemFire Enterprise event handling framework. The CQListener interface is similar to *Application Plug-Ins* on page 85 and *Application Callback Interfaces* on page 110.
- **Complete integration with the client/server architecture**— CQ functionality uses existing server-to-client messaging mechanisms to send events. All tuning of your server-to-client messaging also tunes the messaging of your CQ events. If your system is configured for high availability then your CQs are highly available, with seamless failover provided in case of server failure (see *High Availability for Client-to-Server Communication* on page 152). If your clients are durable, you can also define any of your CQs as durable (see *Durable Client Messaging* on page 154).
- **Interest criteria based on data values**—CQ queries are run against the region’s entry values. Compare this to register interest by reviewing *Registering Interest for Entries* on page 46.
- **Active query execution**—Once initialized, the queries only operate on new events instead of on the entire region data set. Events that change the query result are sent to the client immediately.

In this chapter:

10.1 How Continuous Querying Works

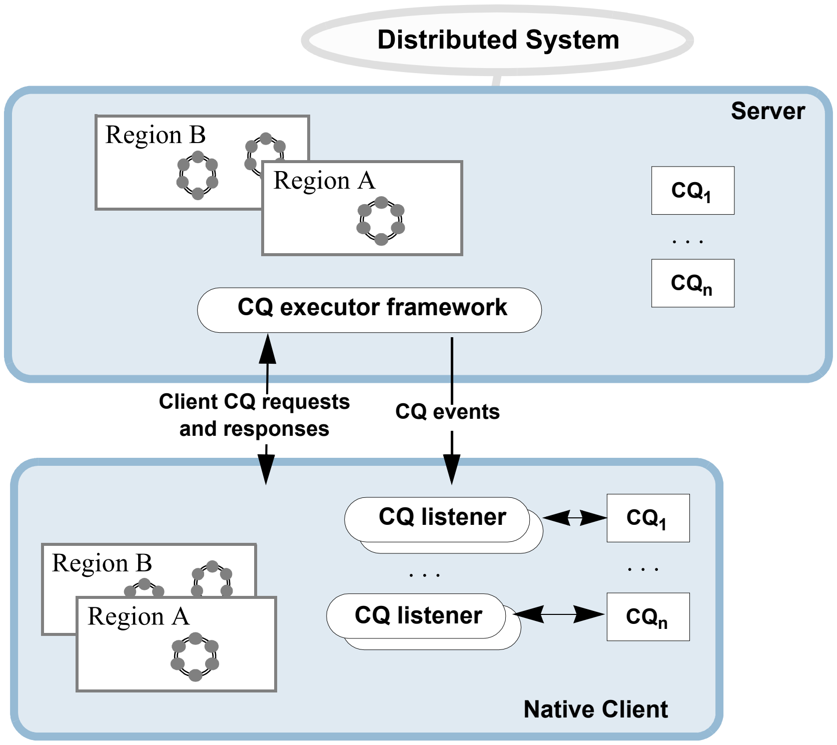
Continuous querying allows you to subscribe to server-side events using SQL-type query filtering. With continuous querying (CQ), the native client sends a query to the server side for execution and receives the events that satisfy the criteria. For example, in a region storing stock market trade orders, you can retrieve all orders over a certain price by running a CQ with a query like this:

```
SELECT * FROM /tradeOrder t WHERE t.price > 100.00
```

When the CQ is running, the server sends the client all new events that affect the results of the query. On the native client side, listeners programmed by you receive and process incoming events. For the example query on `/tradeOrder`, you might program a listener to push events to a GUI where higher-priced orders are displayed. CQ event delivery uses the client/server subscription framework described in *Client to Server Connection Process* on page 72.

Clients can execute any number of CQs, with each CQ given any number of listeners. This figure shows the logical architecture of continuous querying.

Figure 10.1 Logical Architecture of Continuous Querying



CQs do not update the native client region. This is in contrast to other server-to-client messaging, like the updates sent to satisfy interest registration and responses to `get` requests from the client. CQs serve as notification tools for the CQ listeners, which can be programmed in any way your application requires.

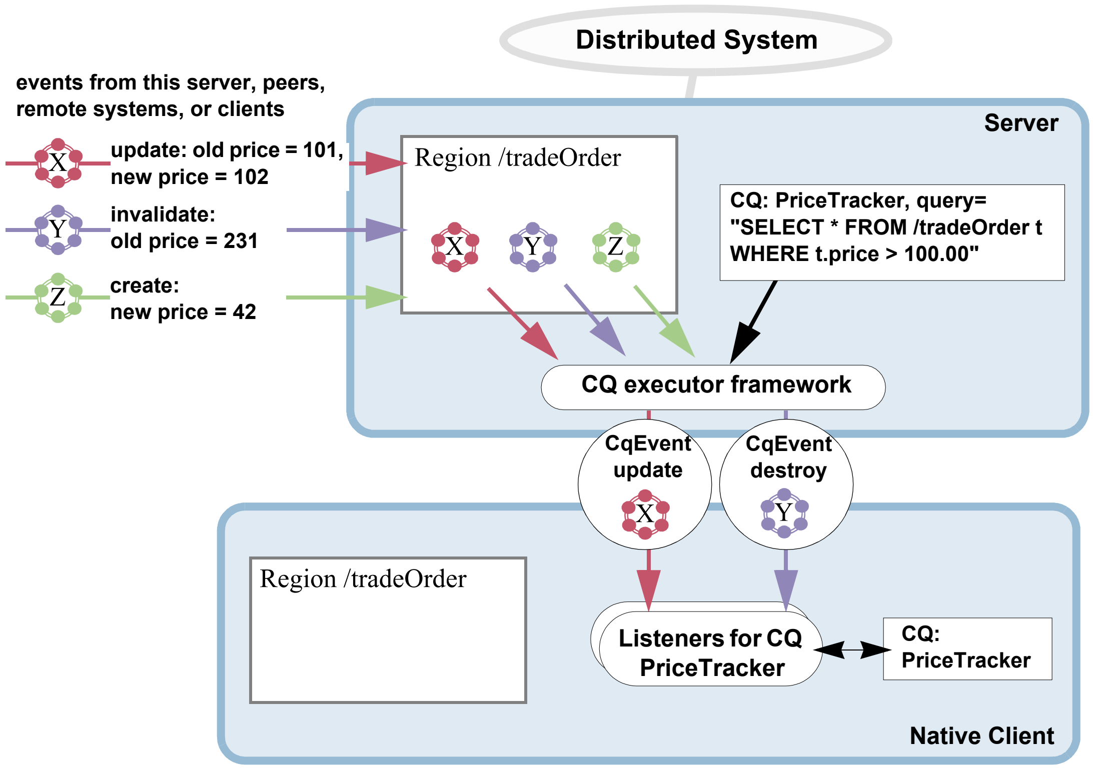
When a CQ is running against a server region, each entry event is evaluated against the CQ query by the thread that updates the server cache. If either the old or the new entry value satisfies the query, the thread puts a `CqEvent` in the client's queue. The `CqEvent` contains information from the original cache event, plus information specific to the CQ's execution. Once received by the client, the `CqEvent` is passed to the `onEvent` method of all `CqListeners` defined for the CQ.

The following figure shows the typical CQ data flow when entries are updated in the server cache. The steps are:

1. Entry events come to the server's cache from any source: the server or its peers, distribution from remote sites, or updates from a client.
2. For each event, the server's CQ executor framework checks for a match with the CQs it has running.
3. If the old or new entry value satisfies a CQ query, a CQ event is sent to the CQ's listeners on the client side. Each listener for the CQ gets the event. In the following figure:
   - Both the new and old prices for entry X satisfy the CQ query, so that event is sent indicating an update to the query results.
   - The old price for entry Y satisfied the query, so it was part of the query results. The invalidation of entry Y makes it not satisfy the query. Because of this, the event is sent indicating that it is destroyed in the query results.
   - The price for the newly created entry Z does not satisfy the query, so no event is sent.

   The region operations do not translate directly to specific query operations, and the query operations do not specifically describe the region events. Instead, each query operation describes how its corresponding region event affects the query results. For more information on this, see The CqEvent Object on page 230.

Figure 10.2 Data Flow of a Continuous Query


Distributed System
events from this server, peers, remote systems, or clients
X
update: old price = 101, new price = 102
Y
invalidate: old price = 231
Z
create: new price = 42
Server
Region /tradeOrder
CQ: PriceTracker, query= "SELECT * FROM /tradeOrder t WHERE t.price > 100.00"
CQ executor framework
CqEvent update
CqEvent destroy
Region /tradeOrder
Listeners for CQ PriceTracker
CQ: PriceTracker
Native Client

10.2 Configuring for Continuous Querying

The continuous query (CQ) functionality requires standard client/server distributed system and cache configuration settings. These are all of the things to consider:

- The client region must use a pool with `subscription-enabled` set to `true`.
- If you want your CQs to be highly available, configure your servers for high availability as described in the *GemFire Enterprise Developer's Guide*. When your servers are highly available, CQs are registered on primary and secondary servers and server failover is performed without any interruption to CQ messaging. CQ events messaging uses the same queues used for server-to-client messaging.

  When CQ is used with high availability, the overhead for CQs is higher than for the key-based interest list registration. CQs are executed on the primary and all secondary servers, so they require more overall server processing.

- If you want your CQs to be durable, configure your native clients for durable messaging. When your clients are durable, you can create durable CQs whose events are maintained during client disconnects and replayed for the client when it reconnects. The process and data flow particular to durable CQs is described in *Durable Client Messaging* on page 154.

10.3 The Native Client CQ API

The GemFire Enterprise native client API allows your clients to create and manage CQs. The server side does not have an API. This section lists the primary native client API for CQ management. For complete information on the classes and interfaces described here, see the online API documentation.

Only C# versions of CQ API interfaces, classes, and methods are shown throughout the text in this chapter (example: `CqQuery.execute`*). The code examples demonstrate both C++ and C# versions.*

Gemstone::GemFire::Cache

- **QueryService**—This interface provides methods to:
  - create a new CQ and specify whether it is durable (available for durable clients)
  - execute a CQ with or without an initial result
  - list all the CQs registered by the client
  - close and stop CQs
  - get a handle on `CqStatistics` for the client

  The `QueryService` *CQ methods are run against the server cache.*

  The `QueryService` may be obtained from the cache or from a pool.
- **CqQuery**—This interface provides methods for managing a continuous query once it is created through the `QueryService`. This is the interface that is normally used to begin and end CQ execution and to retrieve other CQ-related objects such as the CQ attributes, CQ statistics, and the CQ state.
- **CqListener**—This application plug-in interface is programmed by you to handle continuous query events after they occur.
- **CqEvent**—This interface provides all the information sent from the server about the CQ event, which is passed to the CQ's `CqListener` methods.
- **CqState**—This interface gives information on the state of a `CqQuery`. It is provided by the `getState` method of the `CqQuery` instance.
- **CqAttributes, CqAttributesFactory, CqAttributesMutator**—These interfaces allow you to manage CQ attributes. The attributes are composed of `CqListener` plug-in specifications.
- **CqStatistics, CqServiceStatistics**—These interfaces allow you to access statistics information for a single CQ and for the query service's management of CQs as a whole. For details on statistics, see *Statistics API* on page 86.

10.4 State and Life Cycle of a CQ

A CQ has three possible states that can be accessed from the client by calling `CqQuery.getState`.

- **`STOPPED`**—The CQ has been created but not yet executed, or it has been explicitly stopped from executing. The stopped CQ uses system resources. The CQ can be started or restarted by calling the `execute` method on `CqQuery`.
- **`RUNNING`**—The CQ is being executed on the server for all events in the region referenced by the query. Results are sent to all client listeners associated with the `CqQuery`.
- **`CLOSED`**—The CQ is closed and is not using system resources. Invoking an execute or stop method on closed `CqQuery` throws an exception.

A CQ life cycle usually flows like this:

1. The client creates the CQ. This sets up everything for running the query and provides the client with a `CqQuery` object, but does not execute the CQ. At this point, the query is in a `STOPPED` state, ready to be closed or run.
2. The client runs the CQ with an API call to one of the `CqQuery` execute* methods. This puts the query into a `RUNNING` state on the client and on the server.
3. The CQ is closed by a client call to `CqQuery.close`. This de-allocates all resources in use for the CQ on the client and server. At this point, the cycle could begin again with the creation of a new `CqQuery` instance.

10.5 Implementing Continuous Queries

You can specify CQs for any of your client regions. To implement a CQ, follow these basic steps:

1. Make sure your system is configured properly to run CQs. See the configuration guidelines in *Configuring for Continuous Querying* on page 224.
2. Decide what data to track on the server and write your queries. Use your criteria for tracking data changes to write your CQ queries. For information on this, see *Writing the Continuous Query*.
3. Determine how to handle the CQ events on the client and write your listeners.

   Each of your CQs can have any number of listeners. In addition to your core CQ listeners, you might have listeners that you use for all CQs to track statistics or other general information. For information on writing your listeners, see *Writing the CQ Listener* on page 228.
4. Write the client code to put the queries and their listeners into named CQ queries and execute the queries. Make sure to close the queries if your client no longer needs them and when the client exits. For information and examples, see *Running the Continuous Query Code* on page 230.
5. Write any code you need for monitoring and managing your CQ query. For information on the tools available, see *Managing Your CQs* on page 233.
6. Run your code, monitor and tune as needed.

Writing the Continuous Query

Each CQ uses a query and any number of listeners. The query filters the events on the server and the listener handles the events that make it through the query filter. With the query and the listener in hand, you can create and execute your query through the API.

Query Syntax

This is the basic syntax for the CQ query:

```
SELECT * FROM /fullRegionPath [iterator] [WHERE clause]
```

The CQ query must satisfy the standard GemFire Enterprise native client querying specifications described in *The Query Language* on page 189. It also must satisfy these restrictions:

- The `FROM` clause must contain only a single region specification, with optional `iterator` variable.
- The query must be a `SELECT` expression only, preceded by zero or more `IMPORT` statements. This means the query cannot be a statement like `/tradeOrder.name` or `(SELECT * from /tradeOrder).size`.
- The CQ query cannot use:
  - Cross region joins
  - Drill-downs into nested collections
  - `DISTINCT`
  - Projections
  - Bind parameters

Queries not meeting these constraints generate an `UnsupportedOperationException` from the `QueryService` `newCq` method.

Writing the CQ Listener

The following two C++ and C# .NET examples show how you might program a simple `CqListener` to update a display screen based on the CQ events it receives. The listener retrieves the `queryOperation` and entry key and value from the `CqEvent`, then updates the screen according to the operation type provided in `queryOperation`.

Example 10.1 CqListener Implementation (C++)

```
// CqListener class
class TradeEventListener : public CqListener {
   public:
  void onEvent(const CqEvent& cqEvent) {
    // Operation associated with the query op
    CqOperation::CqOperationType queryOperation =
    cqEvent.getQueryOperation();
    // key and new value from the event
    CacheableKeyPtr key = cqEvent.getKey();
    TradeOrderPtr tradeOrder =
    dynCast<TradeOrderPtr>(cqEvent.getNewValue());
    if (queryOperation==CqOperation::OP_TYPE_UPDATE) {
      // update data on the screen for the trade order
      . . .
    }
    else if (queryOperation==CqOperation::OP_TYPE_CREATE) {
      // add the trade order to the screen
      . . .
    }
    else if (queryOperation==CqOperation::OP_TYPE_DESTROY) {
      // remove the trade order from the screen
      . . .
    }
  }
  void onError(const CqEvent& cqEvent) {
    // handle the error
  }
  void close() {
    // close the output screen for the trades
    . . .
  }
}
```

Example 10.2 CqListener Implementation (C#)

```
// CqListener class
public class TradeEventListener : ICqListener {
   public void onEvent(CqEvent cqEvent) {
      // Operation associated with the query op
      CqOperationType queryOperation = cqEvent.getQueryOperation();
      // key and new value from the event
      ICacheableKey key = cqEvent.getKey();
      CacheableString keyStr = key as CacheableString;
      IGFSerializable val = cqEvent.getNewValue();
      TradeOrder tradeOrder = val as TradeOrder;
      if (queryOperation==CqOperationType.OP_TYPE_UPDATE) {
         // update data on the screen for the trade order
         // . . .
         }
      else if (queryOperation== CqOperationType.OP_TYPE_CREATE) {
         // add the trade order to the screen
         // . . .
         }
      else if (queryOperation== CqOperationType.OP_TYPE_DESTROY) {
         // remove the trade order from the screen
         // . . .
      }
   }
   public void onError(CqEvent cqEvent) {
      // handle the error
   }
   // From CacheCallback
   public void close() {
      // close the output screen for the trades
      // . . .
   }
}
```

CQ events do not change your client cache. They are provided as an event service only. This allows you to have any collection of CQs without storing large amounts of data in your regions. If you need to persist information from CQ events, program your listener to store the information where it makes the most sense for your application.

Be very careful if you choose to update your cache from your `CqListener`*. If your listener updates the region that is queried in its own CQ, the update may be forwarded to the server. If the update on the server satisfies the same CQ, it may be returned to the same listener that did the update, which could put your application into an infinite loop. This same scenario could be played out with multiple regions and multiple CQs if the listeners are programmed to update each other's regions.*

The CqEvent Object

The `CqEvent` contains this information:

- Entry key and new value.
- Base operation that triggered the CQ event in the server.
- `CqQuery` object associated with this CQ event.
- Query operation associated with this CQ event. This operation describes the change affected to the query results by the cache event. Possible values are
  - `CREATE`, which corresponds to the standard database `INSERT` operation
  - `UPDATE`
  - `DESTROY`, which corresponds to the standard database `DELETE` operation

This table describes the query operation based on whether the old and new entry values in the region entry event satisfy the query criteria.

Table 10.1 Query Operation Based on Old and New Entry Values

Old Entry Value	New Entry Value	Query Operation
No value or value does not satisfy the query criteria	No value (operation is invalidate or destroy) or value does not satisfy the query	N/A - no event
	Value satisfies the query	create
Value satisfies the query	No value (operation is invalidate or destroy) or value does not satisfy the query	destroy
	Value satisfies the query	update

You can use the query operation to decide what to do with the `CqEvent` in your listeners. For example, a `CqListener` that displays query results on screen might stop displaying the entry, start displaying the entry, or update the entry display depending on the query operation.

When an Error Occurs in a Running CQ

When an error occurs in CQ execution on the server, specific information on the error itself is stored in the server's log file. An exception is passed to the client, then the client throws an exception.

The server log will contain an error with `Error while processing CQ`, like this:

```
[error 2007/12/18 12:03:18.903 PST gemfire1 <RMI TCP Connection(2)-
10.80.10.91> tid=0x18] Error while processing CQ on the event, key :
key-1, CqName :testCQEvents_0, ClientId :
identity(carlos(3249):52623/35391,connection=1,durableAttributes=null)
Error :No public attribute named 'ID' was found in class java.lang.Object
```

Errors in CQ execution are usually caused by data errors, like invalid object types that are stored in the server region. In this case, the query is trying to read into an object of type `Portfolio` for an entry where an empty object has been stored. You can avoid these types of errors by placing constraints on the region entries, or by otherwise controlling the types of objects stored in your server regions.

Running the Continuous Query Code

Create your CQ from an instance of the `QueryService`. Once created, the CQ is maintained primarily through the `CqQuery` interface. The following two C++ and C# examples show the basic calls in the CQ life cycle.

Example 10.3 CQ Creation, Execution, and Close (C++)

```
// Get cache and qrySvcPtr - refs to local cache and QueryService
// Create client /tradeOrder region configured to talk to the server
// Create CqAttribute using CqAttributeFactory
CqAttributesFactory cqf;
// Create a listener and add it to the CQ attributes
//callback defined below
CqListenerPtr tradeEventListener (new TradeEventListener());
QueryServicePtr qrySvcPtr = cachePtr->getQueryService();"
cqf.addCqListener(tradeEventListener);
CqAttributesPtr cqa = cqf.create();
// Name of the CQ and its query
char* cqName = "priceTracker";
char* queryStr = "SELECT * FROM /tradeOrder t where t.price > 100.00";
// Create the CqQuery
CqQueryPtr priceTracker = qrySvcPtr->newCq(cqName, queryStr, cqa); try {
  // Execute CQ
  priceTracker->execute();
} catch (Exception& ex){
  ex.printStackTrace();
}
// Now the CQ is running on the server, sending CqEvents to the listener
. . .
}
// End of life for the CQ - clear up resources by closing
priceTracker->close()
```

Example 10.4 CQ Creation, Execution, and Close (C#)

```
// Get cache and queryService - refs to local cache and QueryService
// Create client /tradeOrder region configured to talk to the server
// Create CqAttribute using CqAttributeFactory
CqAttributesFactory cqf = new CqAttributesFactory();
// Create a listener and add it to the CQ attributes
//callback defined below
ICqListener tradeEventListener = new TradeEventListener();
cqf.addCqListener(tradeEventListener);
CqAttributes cqa = cqf.create();
// Name of the CQ and its query
String cqName = "priceTracker ";
String queryStr = "SELECT * FROM /tradeOrder t where t.price >100.00 ";
QueryService queryService = cache.GetQueryService();
// Create the CqQuery
CqQuery priceTracker = queryService.newCq(cqName, queryStr, cqa, true);
try {
  // Execute CQ
  priceTracker.execute();
  }catch (Exception ex){
  //handle exception
  }
// Now the CQ is running on the server, sending CqEvents to the listener
// . . .
}
// End of life for the CQ - clear up resources by closing
priceTracker.close();
```

CQ Execution Options

CQ execution can be done with or without an initial result set by calling `CqQuery.Execute` or `CqQuery.ExecuteWithInitialResults`. The initial `SelectResults` returned from `ExecuteWithInitialResults` is the same as the one you would get if you ran the query ad hoc by calling `QueryService.NewQuery.Execute` on the server cache, but with the key added.

If you are managing a data set from the CQ results, you can initialize the set by iterating over the result set and then updating it from your listeners as events arrive. For example, you might populate a new screen with initial results and then update the screen from a listener.

Just as with the standalone query, the initial results represents a possibly moving snapshot of the cache. If there are updates to the server region while the result set is being created, the result set and the subsequent event-by-event CQ query execution might miss some events.

Managing Your CQs

This section discusses how to access and manage your CQs from your client. The calls discussed here are all executed specifically for the calling client. A client cannot access or modify the CQs belonging to another client. For detailed method usage, see the online API documentation.

Accessing Your CQs

You can access a single named CQ, an array of all CQs registered, and an array of all CQs registered in the client using the `QueryService getCq*` methods. You can also access the CQ used to produce a `CqEvent` through the `CqEvent.getCq` method.

Accessing CQ Statistics

CQ runtime statistics are available for the client through the `CqServiceStatistics` and `CqStatistics` interfaces described under *The Native Client CQ API* on page 225. You can get information on the events generated by a specific CQ from the `CqStatistics` object returned by `CqQuery.GetStatistics`. You can get higher-level information about the CQs the client has registered, running, and so on, from the `CqServiceStatistics` object returned by `QueryService.GetCqStatistics`.

For both the client and server, you can access these statistics by loading the statistics archive file into VSD. The optional VSD (Visual Statistics Display) tool can be acquired from GemStone Technical Support. See *Contacting Technical Support* on page 21 for details.

Client statistics are for the single client only. The server's pertain to all clients with CQs on this server.

Modifying CQ Attributes

You can modify the attributes for an existing CQ using the methods provided by `CqQuery.GetCqAttributesMutator`. The attributes consist of a list of listeners.

Executing CQs

Individual CQs are executed using `CqQuery execute*` methods. You can also execute all CQs for the client or for a region through the client `QueryService`. CQs that are running can be stopped or closed.

Stopping CQs

Individual CQs are stopped using the `CqQuery stop` method. You can also stop all CQs for the client through the `QueryService`. Stopped CQs are in the same state as new CQs that have not yet been executed. You can close or execute a stopped CQ.

Closing CQs

Individual CQs are closed using the `CqQuery close` method. You can also close all CQs for the client through the `QueryService`. Closed CQs cannot be executed. CQs are also closed in the following cases:

- The client closes its cache after closing all of its CQs—Closing the client cache closes the `QueryService` and all associated CQs on the client and server.
- The client disconnects from its server—This might be because of a network outage or some other failure. When a client disconnects, all CQs created by the client are removed from the server and put into a `CLOSED` state on the client.
- The server region is destroyed—When a server region is destroyed, all associated CQs are also cleaned up on the server and the region `destroy` event is sent to the client. On the client, the `CqListener.Close` method is called for all CQs on the region.

Chapter 11

Using Connection Pools

The connection pool API supports connecting to servers through server locators or directly connecting to servers. In a distributed system, servers can be added or removed and their capacity to service new client connections may vary. The locators continuously monitor server availability and server load information and provide clients with the connection information of the server with the least load at any given time. Locators provide clients with dynamic server discovery and server load balancing. Clients are configured with locator information, and depend on the locators for server connectivity.

Server locators provide these main features:

- Automated Discovery of Servers and Locators—Adding and removing servers or locators is made easy as each client does not require a list of servers to be configured at the time of pool creation.
- Client Load Rebalancing—Spreading the client load over servers in a distributed system is made efficient. Server locators give clients dynamic server information and provide server load rebalancing after servers depart or join the system.
- High Availability—When a client/server connection receives an exception, the connection is automatically failed over to another available connection in the pool. Redundancy is also provided for client subscriptions.

You can alternatively configure a pool statically with a list of endpoints. When the pools are statically configured, a round-robin load balancing policy is used to distribute connections across the servers.

In this chapter:

11.1 How Client Load Balancing Works

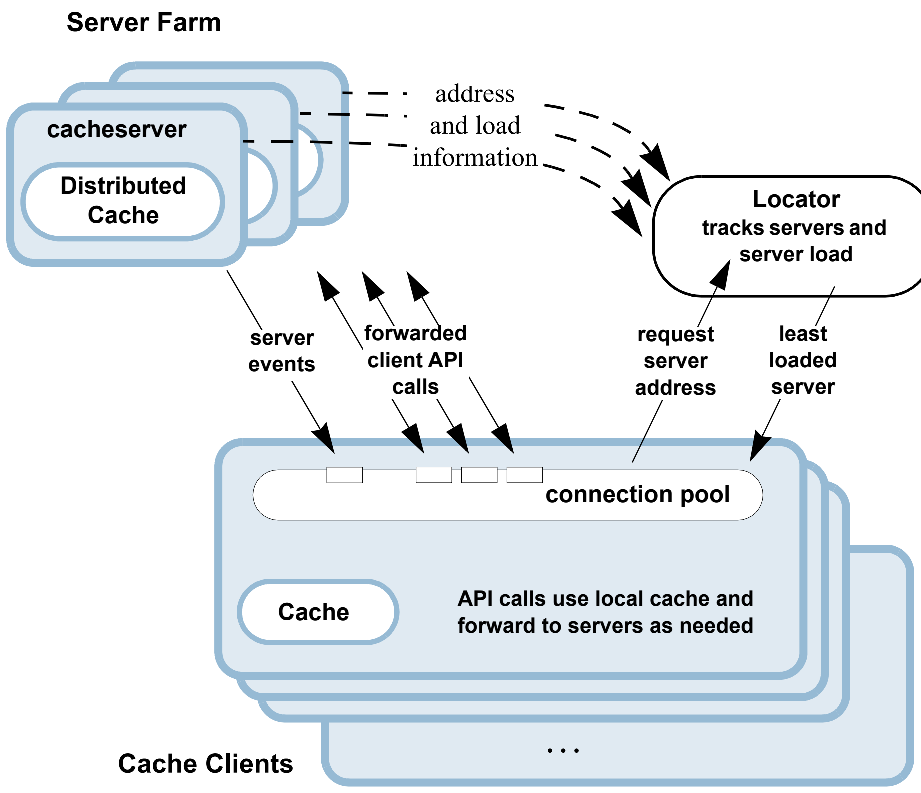
The client is configured with a list of server locators and consults a server locator to request a connection to a server in the distributed system.

Clients contain connection pools. Each region is associated with a connection pool using a region attribute and operations on the region use connections from the respective pools. The server connectivity options are specified in the connection pool configuration. Each pool has a minimum and maximum number of connections.

Each cache operation that requires server connectivity obtains a connection from the pool for the server group that the operation affects, performs the operation using the connection, and returns the connection to the pool. If the pool size is less than the maximum number of connections and all connections are in use, the connection pool creates a new connection and returns it. If the pool is at the maximum number of connections, that thread blocks until a connection becomes available or a `free-connection-timeout` occurs. If a `free-connection-timeout` occurs, an `AllConnectionsInUse` exception is thrown.

The connection pool has a configurable time-out period that is used to expire idle connections. The idle connections are expired until the pool has the minimum number of connections. A monitoring thread expires idle connections, adds sufficient connections to bring up the count to minimum, closes connections whose lifetime has been exceeded. See load-conditioning-interval and idle-timeout. A separate thread (ping) tests each connected endpoint for its status and if the endpoint is not reachable, the thread closes all connections that have been made to the endpoint. See ping-interval.

Figure 11.1 Logical Architecture of Client/Server Connections


Server Farm
cacheserver
Distributed Cache
address and load information
Locator
tracks servers and server load
server events
forwarded client API calls
request server address
least loaded server
connection pool
Cache
API calls use local cache and forward to servers as needed
. . .
Cache Clients

When a connection receives an exception, the operation is failed over to another connection from the pool. The failover mechanism obtains the endpoint to failover to from the locator or from the specified endpoint list in the pool.

Discovering Locators Dynamically

Like servers, locators can be discovered dynamically. A background thread periodically queries the locator for any other locators joining the distributed system. In order to balance load across all the locators, locators are shuffled before pushing in the list.

However, If locator A (to which the client is connected) goes down before it could discover locator B, the locator B is never discovered even though it is alive and client receives a `NoLocatorsAvailable` exception.

Thread Local Connection—As the name suggests, one connection is attached to every application thread that is `local` to the respective thread.

In this case, to perform any cache operation the client is not required to obtain a connection from pool. Instead the thread local connection of the client is used.

A thread local connection can be released by invoking the `Pool::releaseThreadLocalConnection()` method. The released connection is returned to the pool. If the number of threads is larger than the number of `max-connections`, the client throws an `AllConnectionsInUseException` after the `free-connection-timeout` lapses, unless the `Pool::releaseThreadLocalConnection() method` is used judiciously.

If a connection expires or the server goes down on which the connection was established, a thread local connection is immediately replaced with a good connection obtained from the pool.

11.2 Configuring Pools for Servers or Locators

The connection pools require standard client/server distributed system and cache configuration settings. In addition, the following settings must be configured for the locator, server, and the pool elements:

- Locator—Specify the host and port where a server locator is listening.
- Server—Specify the host and port where a server is listening.
- Pool—Specifies a client/server connection pool. The following attributes can be configured:

Table 11.1 Pool Attributes

Attribute Name	Description	Default
`free-connection-timeout`	The amount of time, in milliseconds (ms), that the client waits for a free connection if `max-connections` limit is configured and all of the connections are in use.	10000 ms
`idle-timeout`	The amount of time, in milliseconds, to wait for a connection to become idle for load balancing	5000 ms
`load-conditioning-interval`	The interval in which the pool checks to see if a connection to a specific server should be moved to a different server to improve the load balance.	300000 ms (5 minutes)
`max-connections`	The maximum number of connections that the pool can create. If all of the connections are in use, an operation requiring a client to server connection blocks until a connection is available or the `free-connection-timeout` is reached. If set to -1, there is no maximum. The setting must indicate a cap greater than `min-connections`. *If you need to use this to cap your pool connections, you should disable the pool attribute* `pr-single-hop-enabled`. *Leaving single hop enabled can increase thrashing and lower performance.*	-1
`min-connections`	The number of connections that must be created initially.	5
`name`	The pool name.	
`ping-interval`	How often to communicate with the server to show the client is alive, set in milliseconds. Pings are only sent when the `ping-interval` elapses between normal client messages. This must be set lower than the server's `maximum-time-between-pings`.	10000 ms
`pr-single-hop-enabled`	Setting used to for single-hop access to partitioned region data in the servers for some data operations. See *PartitionResolver* on page 64. See note in `thread-local-connections` (page 239).	True
`read-timeout`	The amount of time, in milliseconds, to wait for a response from a server before the connection times out.	10000
`retry-attempts`	The number of times to retry an operation after a time-out or exception for high availability. If set to -1, the pool tries every available server once until it succeeds or has tried all servers.	-1
`server-group`	The server group to select connections from. If not specified, the global group of all connected servers is used.	empty

Attribute Name	Description	Default
`socket-buffer-size`	Size of the socket buffer, in bytes, on each connection established.	32768
`statistic-interval`	The default frequency, in milliseconds, with which the client statistics are sent to the server. A value of `-1` indicates that the statistics are not sent to the server.	-1
`subscription-ack-interval`	The amount of time, in milliseconds, to wait before sending an acknowledgement to the server about events received from the subscriptions.	500
`subscription-enabled`	Whether to establish a server to client subscription.	False
`subscription-message-tracking-timeout`	The amount of time, in milliseconds, for which messages sent from a server to a client are tracked. The tracking is done to minimize duplicate events.	90000
`subscription-redundancy`	The redundancy for servers that contain subscriptions established by the client. A value of `-1` causes all available servers in the specified group to be made redundant.	0
`thread-local-connections`	Whether the connections must have affinity to the thread that last used them. *To set this to* `true`*, also set* `pr-single-hop-enabled` (page 238) *to* `false`*. A* `true` *value in* `pr-single-hop-enabled` *automatically assigns a* `false` *value to* `thread-local-connections`.	False

The locator, server, and pool settings can be configured declaratively in `client` XML or programmatically through the `PoolFactory` method.

Create an instance of `PoolFactory` *through* `PoolManager`.

This example shows declarative pool configuration:

Example 11.1 Configuring a Pool

```
<pool free-connection-timeout="12345" idle-timeout="5555"
       load-conditioning-interval="23456"
       max-connections="7" min-connections="3"
       name="test_pool_1" ping-interval="12345"
       read-timeout="23456" retry-attempts="3" server-group="ServerGroup1"
       socket-buffer-size="32768" statistic-interval="10123"
       subscription-ack-interval="567" subscription-enabled="true"
       subscription-message-tracking-timeout="900123"
       subscription-redundancy="0" thread-local-connections="true">
    <locator host="localhost" port="34756"/>
</pool>
```

11.3 The Native Client Pool API

The GemFire Enterprise native client API allows your clients to create and manage connection pools. The server side does not have an API. This section lists the primary native client API for pool management. For complete information on the classes and interfaces described here, see the online API documentation.

Only C# versions of Pool API interfaces, classes, and methods are shown throughout the text in this chapter (example: `Pool.GetQueryService()`*). The code examples demonstrate both C++ and C# versions.*

Gemstone::GemFire::Cache

- **Pool**—This interface provides the API to retrieve pool attributes.
- **PoolFactory**—This interface provides the API to configure pool attributes.
- **PoolManager**—This interface provides API to create a `PoolFactory` object and to find the pool objects.
- The `AttributesFactory` class has a new method `setPoolname` which assigns a pool to a region. Operations performed on the configured region use connections from the pool.

  A region can have a pool attached to it. A pool may have multiple regions attached to it.

11.4 Working With Pools

A pool can be configured as:

- Locator(s) or
- List of servers

You can configure the pool properties either declaratively through `client` XML or programmatically through the `PoolFactory` API.

Subscription Properties

Each connection pool has a single subscription connection which could be to any server which matches the requirements of the connection pool. When a client registers interest for a region, if the connection pool does not already have a subscription channel, the connection pool sends a message to the server locator and the server locator chooses servers to host the queue and return those server names to the client. The client then contacts the chosen servers and asks them to create the queue. The client maintains at least one connection with each server hosting a queue - if the server does not detect any connections from a non-durable client, it drops the client queue and closes all artifacts for the client. For information about durable client subscriptions, see *Durable Client Messaging* on page 154.

Requesting a Subscription Region Queue

The client to server locator request is a short lived TCP request. The client sends a message with:

- The client ID
- (Optional) target server group
- Number of redundant copies
- Servers to exclude from the results. This is used if the client was unable to connect to a server and needs to request a new one.

The server locator responds with a list of servers. The client is responsible for contacting the primary and secondaries and asking them to host the queue.

For durable subscriptions, the server locator must be able to locate the servers that host the queues for the durable client. When a durable client sends a request, the server locator queries all the available bridge servers to see if they are hosting the subscription region queue for the durable client. If the server is located, the client is connected to the server hosting the subscription region queue.

Running the Connection Pool Code

The following examples demonstrate a simple procedure to create a pool factory and then create a pool instance in C++ and C#. They also help you to execute a query. The following examples create a pool with locators. Ensure that you create a pool with locators or endpoints, but not both. The first example demonstrates creating a pool by adding locators. The second example demonstrates creating a pool by adding servers. For more information, see the example in the QuickStart Guide.

Example 11.2 Connection Pool Creation and Execution Using C++

```
PropertiesPtr prptr = Properties::create();
systemPtr = CacheFactory::createCacheFactory(prptr);

cachePtr = systemPtr->create();
PoolFactoryPtr poolFacPtr = PoolManager::createFactory();
//to create pool add either endpoints or add locators or servers
//pool with endpoint, adding to pool factory
//poolFacPtr->addServer("localhost", 12345 /*port number*/);
//pool with locator, adding to pool factory
poolFacPtr->addLocator("localhost", 34756 /*port number*/);
PoolPtr pptr = NULLPTR;
if ((PoolManager::find("examplePool")) == NULLPTR) {// Pool does not exist
with the same name.
   pptr = poolFacPtr->create("examplePool");
}
RegionFactoryPtr regionFactory =
   cachePtr->createRegionFactory(CACHING_PROXY);
regionPtr = regionFactory
   ->setPoolName("examplePool")
   ->create("regionName");
QueryServicePtr qs = cachePtr->getQueryService("examplePool");
```

Example 11.3 Connection Pool Creation and Execution Using (C#)

```
Properties prop = Properties.Create();
CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(prop);

Cache cache = cacheFactory.Create();

PoolFactory poolFact = PoolManager.CreateFactory();
//to create pool add either endpoints or add locators
//pool with endpoint, adding to pool factory.
poolFact.AddServer("localhost", 40404 /*port number*/);
//pool with locator, adding to pool factory
//poolFact.AddLocator("hostname", 15000 /*port number*/);
Pool pool = null;
if (PoolManager.Find("poolName") == null) {
   pool = poolFact.Create("poolName");
}
int loadConfigInterval = pool.LoadConditioningInterval;
RegionFactory regionFactory =
   cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
Region region = regionFactory.SetPoolName("poolName").Create("regionName");
QueryService qs = cache.GetQueryService("poolName");
```

Chapter 12 Function Execution

Using the GemFire Enterprise® function execution service, you can execute application functions on a single server member, in parallel on a subset of server members, or in parallel on all server members of a distributed system. Achieving linear scalability is predicated upon being able to horizontally partition the application data such that concurrent operations by distributed applications can be done independently across partitions. In other words, if the application requirements for transactions can be restricted to a single partition, and all data required for the transaction can be colocated to a single server member or a small subset of server members, then true parallelism can be achieved by vectoring the concurrent accessors to the ever-growing number of partitions.

Most scalable enterprise applications grow in terms of data volume, where the number of data items managed rather than each item grows over time. If the above logic holds (especially true for OLTP class applications), then we can derive sizable benefits by routing the data dependent application code to the fabric member hosting the data. The term we use to describe this routing of application code to the data of interest is aptly called *data-aware function routing*, or *behavior routing*.

Function Execution can be used only along with the pool functionality. For more information about the pool API, see *Using Connection Pools* on page 235.

In this chapter:

Only C++ versions of Function Execution API interfaces, classes, and methods are shown throughout the text in this chapter (like `FunctionService::onRegion`). The code examples show C++ and C#.

12.1 How Function Execution Works

Where functions are executed

You can specify the members that run the function or the data set over which to run.

- **Servers**. You can execute the function in a single server or a set of servers, specified by the server pool. To specify data sets for this type of function, pass arguments in to the function.
- **Data set**. For this, you specify a region and possibly a set of keys on which to run.

How functions are executed

1. The calling client application runs the `execute` method on the `Function Execution` object. The object must be already registered on the servers.
2. The execution is sent to all servers where it needs to run. The locations are determined by the `FunctionService on*` method calls, region configuration, and any filters.
3. If the function has results, the result is returned to the `execute` method call in a `ResultCollector` object.
4. The client collects results using the result collector `getResult`.

Highly available functions

Generally, if there is a failure in function execution, the error is returned to the calling application. You can code for high availability for `onRegion` functions that return a result, so the function is automatically retried. For information on setting this up on the server side, see the *GemFire Enterprise Developer's Guide.* To use a highly available function, the client must call the results collector `getResult` method. When there is a failure—execution error or member crash while executing, for example—the system

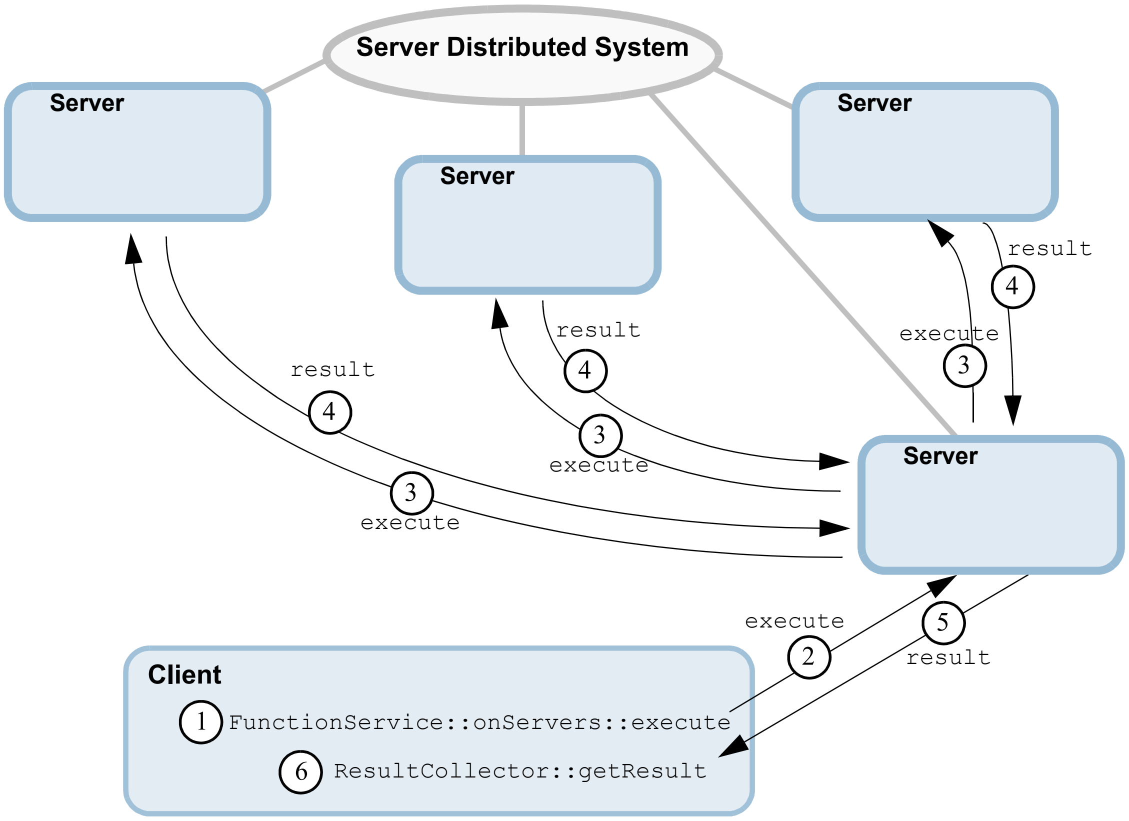
1. Waits for all calls to return
2. Sets a boolean indicating a reexecution is being done
3. Calls the result collector's clearResults method
4. Executes the function

The system retries the execution up to the number specified in the server pool's `retryAttempts` setting. If the function continues to fail, the final exception is returned to the `getResult` method.

Data-independent function execution

This shows the sequence of events for a data-independent function executed against all available servers:

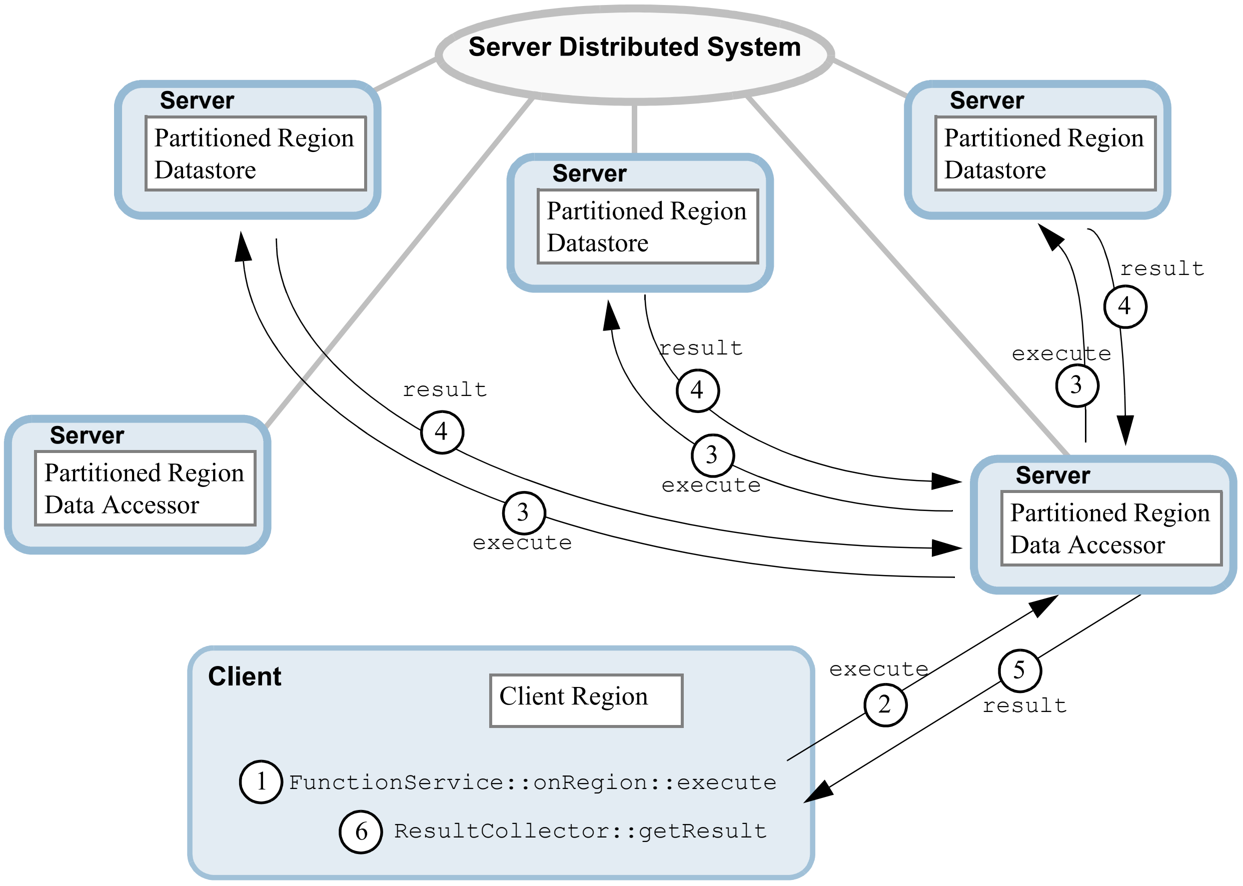
Figure 12.1 Data-Independent Function Invoked from a Client


Server Distributed System
Server
Server
Server
Server
result
4
execute
3
result
4
3
execute
result
4
3
execute
Client
1 FunctionService::onServers::execute
6 ResultCollector::getResult
execute
2
5
result

Data-dependent function execution

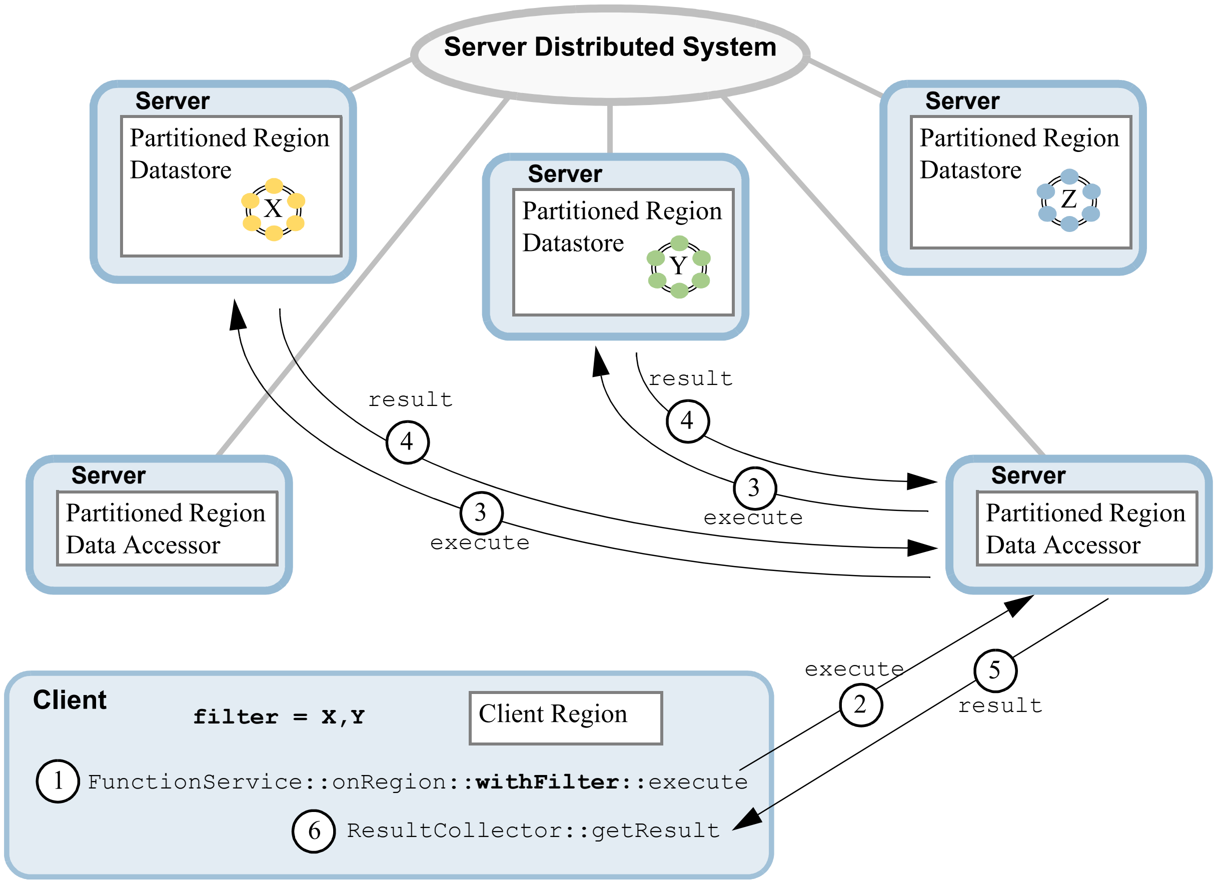
This shows a data-dependent function run by a client. The specified region is connected to the server system, so the function automatically goes there to run against all servers holding data for the region.

Figure 12.2 Data-Dependent Function Invoked from a Client


Server Distributed System
Server
Partitioned Region Datastore
Server
Partitioned Region Datastore
Server
Partitioned Region Datastore
Server
Partitioned Region Data Accessor
Server
Partitioned Region Data Accessor
result
4
execute
3
result
4
execute
3
result
4
execute
3
Client
Client Region
1 FunctionService::onRegion::execute
6 ResultCollector::getResult
execute
2
5
result

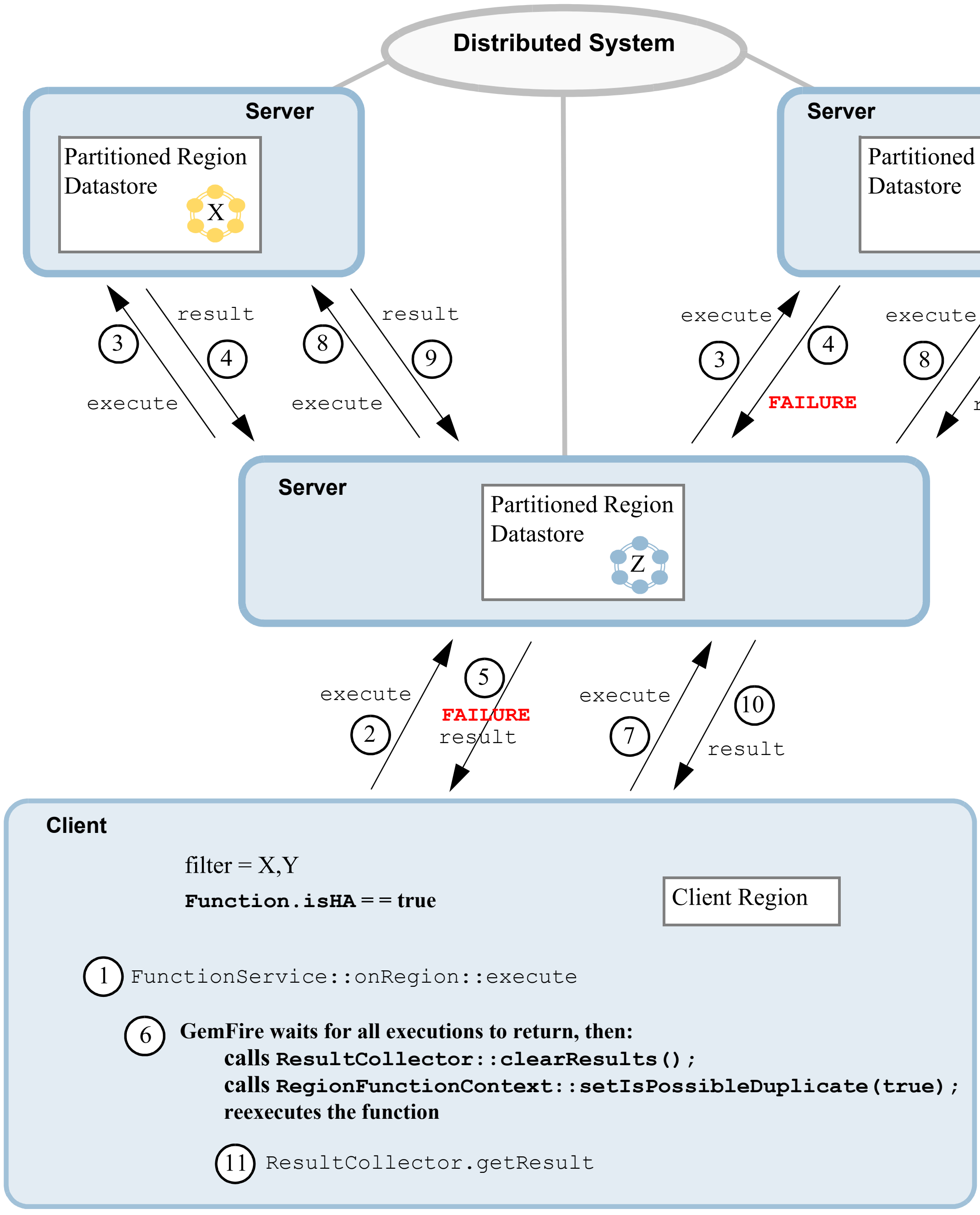
This shows the same data-dependent function with the added specification of a set of keys on which to run. Servers that don't hold any of the keys are left out of the function execution.

Figure 12.3 Data-Dependent Function with Filter Invoked from a Client


Server Distributed System
Server
Partitioned Region Datastore
X
Server
Partitioned Region Datastore
Y
Server
Partitioned Region Datastore
Z
Server
Partitioned Region Data Accessor
Server
Partitioned Region Data Accessor
result
4
3
execute
result
4
3
execute
execute
2
5
result
Client
filter = X,Y
Client Region
1 FunctionService::onRegion::withFilter::execute
6 ResultCollector::getResult

This scenario demonstrates the steps in a call to a highly available function. The call fails the first time on one of the participating servers and is successfully run a second time on all servers.

Figure 12.4 Highly Available Data Dependent Function with Failure on First Execution


Distributed System
Server
Partitioned Region Datastore
X
Server
Partitioned Region Datastore
Y
3 execute
4 result
8 execute
9 result
3 execute
4 FAILURE
8 execute
9 result
Server
Partitioned Region Datastore
Z
2 execute
5 FAILURE result
7 execute
10 result
Client
filter = X,Y
Function.isHA == true
Client Region
1 FunctionService::onRegion::execute
6 GemFire waits for all executions to return, then:
calls ResultCollector::clearResults();
calls RegionFunctionContext::setIsPossibleDuplicate(true);
reexecutes the function
11 ResultCollector.getResult

12.2 Executing Functions in GemFire

This assumes you have your client and server regions defined and you have coded and configured your servers to run your functions. See the function execution information for the server side in the *GemFire Enterprise Developer's Guide*.

All functions must be written in Java.

5. If your function returns results and you need special results handling, code a custom `ResultsCollector` implementation to replace the default provided by GemFire. Use the `Execution::withCollector` method to define your custom collector.
6. Write the application code to run the function and, if the function returns results, to get the results from the collector. For high availability, the function must return results.

Running the function

In every client where you want to execute the function and process the results:

1. Use one of the `FunctionService` `on*` methods to create an `Execution` object. The `on*` methods, `onRegion`, `onServer` and `onServers`, define the highest level where the function is run. If you use `onRegion` you can further narrow your run scope by setting key filters. The function run using `onRegion` is a data dependent function - the others are data-independent functions.

   You can run a data dependent function against custom partitioned and colocated partitioned regions. The steps for setting up the regions and functions in the server are provided in the *GemFire Enterprise Developer's Guide*. From the client, just provide the appropriate key sets to the function call.
2. Use the `Execution` object as needed for additional function configuration. You can:
   - Provide a set of data keys to `withFilter` to narrow the execution scope. This works only for `onRegion` `Execution` objects.
   - Provide function arguments to `withArgs`.
   - Define a custom `ResultCollector` to `withCollector`
3. Call the `Execution` object `execute` method to run the function.
4. To run a function with high availability, call `getResult` from the results collector returned from `execute`. Calling a highly available function without using `getResult` disables the high availability functionality.

Example 12.1 Running a function on a region in C++

```
regPtr0 = initRegion();
ExecutionPtr exc = FunctionService::onRegion(regPtr0);
CacheableVectorPtr routingObj = CacheableVector::create();
char buf[128];
bool getResult = true;

sprintf(buf, "VALUE--%d", 10);
CacheablePtr value(CacheableString::create(buf));

sprintf(buf, "KEY--%d", 100);
CacheableKeyPtr key = CacheableKey::create(buf);
regPtr0->put(key, value);

sprintf(buf, "KEY--%d", 100);
CacheableKeyPtr key1 = CacheableKey::create(buf);
routingObj->push_back(key1);

CacheablePtr args = routingObj;
CacheableVectorPtr executeFunctionResult = exc->withFilter(routingObj)->
       withArgs(args)->execute(func, getResult)->getResult();
```

Example 12.2 Running a function on a region in C#

```
Region rg=<create Region>
IGFSerializable[] routingObj = new IGFSerializable[17];
int j=0;
for(int i=0; i < 34; i++)
{
if(i%2==0) continue;
routingObj[j] = new CacheableString("KEY--"+i);
j++;
}
Execution exc = FunctionService.OnRegion(rg);
CacheableVector args1 = new  CacheableVector();
for(int i=0; i < routingObj.Length; i++)
{
Console.WriteLine("routingObj[{0}]={1}.", i, (routingObj[i] as
CacheableString).Value);
args1.Add(routingObj[i]);
}
IResultCollector rc =  exc.WithArgs(args1).WithFilter(routingObj).Execute(
getFuncName, getResult);
IGFSerializable[] executeFunctionResult = rc.GetResult();
```

Example 12.3 Running a function on a server pool in C++

```
pptr = PoolManager::find(poolName);
ExecutionPtr exc = FunctionService::onServer(cache);
CacheableVectorPtr routingObj = CacheableVector::create();
char buf[128];
bool getResult = true;
sprintf(buf, "VALUE--%d", 10);
CacheablePtr value(CacheableString::create(buf));

sprintf(buf, "KEY--%d", 100);
CacheableKeyPtr key = CacheableKey::create(buf);
regPtr0->put(key, value);

sprintf(buf, "KEY--%d", 100);
CacheableKeyPtr key1 = CacheableKey::create(buf);
routingObj->push_back(key1);

CacheablePtr args = routingObj;
CacheableVectorPtr executeFunctionResult =
   exc->withArgs(args)->execute(func, getResult)->getResult();
```

Example 12.4 Running a function on a server pool in C#

```
Execution exc = FunctionService.OnServer(cache);
CacheableVector args1 = new  CacheableVector();
for(int i=0; i < routingObj.Length; i++)
{
   Console.WriteLine("routingObj[{0}]={1}.", i, (routingObj[i] as
               CacheableString).Value);
   args1.Add(routingObj[i]);
}
IResultCollector rc =  exc.WithArgs(args1).Execute(getFuncIName, getResult);
IGFSerializable[] executeFunctionResult = rc.GetResult();
Console.WriteLine("on one server: result count= {0}.",
               executeFunctionResult.Length);
```

Handling function results

This section applies only to functions that return results.

To program your client to get the results from a function, use the result collector returned from the function execution, like this:

```
ResultCollectorPtr rc = FunctionService::onRegion(region)
                      ->withArgs(args)
                      ->withFilter(keySet)
                      ->withCollector(new MyCustomResultCollector())
                      .execute(Function);
CacheableVectorPtr functionResult = rc.getResult();
```

GemFire provides a default result collector. Its `getResult` methods block until all results are received, then return the full result set.

You can handle the results in a custom manner if you wish. To do this:

1. Write a class that extends `ResultCollector` and code the methods to handle the results as you need. Note that the methods are of two types - one handles data and information from GemFire and populates the results set, while the other returns the compiled results to the calling application.:
   1.1 `addResult` is called by GemFire when results arrive from the `Function` methods. Use `addResult` to add a single result to the `ResultCollector`.

   1.2 `endResults` is called by GemFire to signal the end of all results from the function execution.

   1.3 `getResult` is available to your executing application (the one that calls `Execution.execute`) to retrieve the results. This may block until all results are available.

   1.4 `clearResults` is called by GemFire to clear partial results from the results collector. This is used only for highly available `onRegion` functions where the calling application waits for the results. If the call fails, before GemFire retries the execution, it calls `clearResults` to ready the instance for a clean set of results.
2. Use the `Execution` object in your executing member to call `withCollector`, passing your custom collector, as shown in the example above.

12.3 Solutions and Use Cases

The function execution service provides solutions for the following application use cases:

- An application intending to execute a server-side transaction or carry out data updates using the GemFire distributed locking service.
- An application wanting to initialize some of its components once on each server, which might be used later by executed functions.
- Initialization and startup of a third-party service, such as a messaging service.
- Any arbitrary aggregation operation that requires iteration over local data sets that can be done more efficiently through a single call to the cache server.
- Any kind of external resource provisioning that can be done by executing a function on a server.

Chapter 13 Delta Propagation

In most distributed data management systems, the data stored in the system tends to be created once and then updated frequently. These updates are sent to other members for event propagation, redundancy management, and cache consistency in general. Tracking only the changes in an updated object and sending only the updates, or deltas, mean lower network transmission costs and lower object serialization/deserialization costs. Performance improvements can be significant, especially when changes to an object instance are relatively small compared to the overall size of the instance.

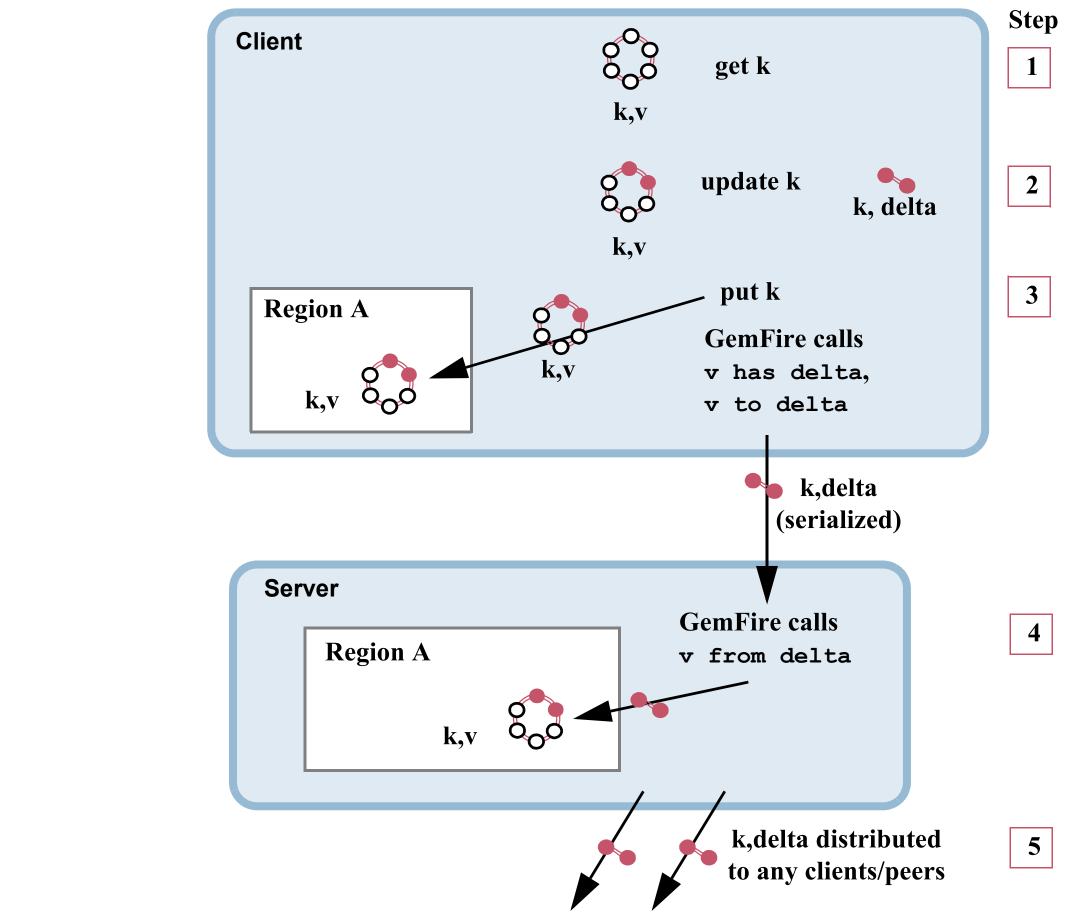
In this chapter:

13.1 How Delta Propagation Works

GemFire propagates object deltas using methods that you program. The methods are in the delta interface, which you implement in your cached objects' classes.

This figure shows delta propagation for a change to an entry with key, `k`, and value object, `v`.

Figure 13.1 Delta propagation



1. **`get` operation**. The `get` works as usual; the cache returns the full entry object from the local cache or, if it isn't available there, from a server cache or from a loader.
2. **update methods**. You need to add code to the object's update methods so that they save delta information for object updates, in addition to the work they were already doing.
3. **`put` operation**. The `put` works as usual in the local cache, using the full value, then calls `hasDelta` to see if there are deltas and `toDelta` to serialize the information.
4. **receipt of delta**. `fromDelta` extracts the delta information that was serialized by `toDelta` and applies it to the object in the local cache. The delta is applied directly to the existing value or to a clone, depending on how you configure it for the region.
5. **additional distributions**. As with full distributions, receiving members forward the delta according to their configurations and connections to other members. In the example, the server would forward the delta to its peers and its other clients as needed. Receiving members do not recreate the delta; `toDelta` is only called in the originating member.

13.2 Supported Topologies and Limitations

General Notes and Limitations

Sometimes `fromDelta` cannot be invoked because there is no object to apply the delta to in the receiving cache. When this happens, the system sends the full value. There are two possible scenarios:

- If the system can determine beforehand that the receiver does not have a local copy, it sends the initial message with the full value. This is possible when regions are configured with no local data storage, like when you are using them to provide data update information to listeners.
- In the less obvious cases, like when an entry has been locally deleted, first the delta is sent, then the receiver requests a full value and that is sent. Whenever the full value is received, any further distributions to the receiver’s peers or clients uses the full value.

GemFire does not propagate deltas for:

- Transactional commit on the server
- The `putAll` operation
- Server VMs running GemFire versions 6.0 and earlier

GemFire clients can always send deltas to the servers, and servers can usually sent deltas to clients. These configurations require the servers to send full values to the clients, instead of deltas:

- When the client's `gemfire.properties` setting `conflate-events` is set to `true`, the servers send full values for all regions.
- When the server region attribute `enable-subscription-conflation` is set to `true` and the client `gemfire.properties` setting `conflate-events` is set to `server`, the servers send full values for the region.
- Servers send full values to client regions that are configured to not cache data—with the `PROXY RegionShortcut` in their region attributes `refid`.

To use the delta propagation feature, all updates on a key in a region must have value types that implement the `Delta` interface. You cannot mix object types for an entry key where some of the types implement delta and some do not. This is because, when a type implementing the delta interface is received for an update, the existing value for the key is cast to a `Delta` type to apply the received delta.

13.3 Delta Propagation API

Delta propagation uses configuration and a simple API to send and receive deltas.

The native client delta propagation API is provided in these classes and interfaces.

Only the C++ class and method names are used for the discussions in this chapter.

.NET - for C#

Your application class must implement:

- `GemStone::GemFire::Cache::IGFDelta`
- `GemStone::GemFire::Cache::IGFSerializable`

`IGFDelta` provides the methods, `HasDelta`, `ToDelta`, and `FromDelta`, which you program to report on, send, and receive deltas for your class.

Additionally, for cloning, your class must implement the standard .NET `IClonable` interface and its `Clone` method.

C++

Your application must publicly derive from:

- `gemfire::Delta`
- `gemfire::Cacheable`

`Delta` provides the methods, `hasDelta`, `toDelta`, `fromDelta`, which you program to report on, send, and receive deltas for your class.

For cloning, use the `clone` method provided in the `Delta` interface.

13.4 Delta Propagation Properties

Delta propagation is configured on the server side as well as client. For information on the server and delta propagation, see the delta propagation chapter in the *GemFire Enterprise Developer's Guide*.

cloning-enabled

A region attributes boolean, configured in the `cache.xml`, that affects how `fromDelta` applies deltas to the local client cache. When `true`, the updates are applied to a clone of the value and then the clone is saved to the cache. When false, the value is modified in place in the cache. The default value is false.

Cloning can be expensive, but it ensures that the new object is fully initialized with the delta before any application code sees it.

When cloning is enabled, by default GemFire does a deep copy of the object, using serialization. You may be able to improve performance by implementing the appropriate `clone` method for your API, making a deep copy of anything to which a delta may be applied. The goal is to significantly reduce the overhead of copying the object while still retaining the isolation needed for your deltas.

Without cloning:

- It is possible for application code to read the entry value as it is being modified, possibly seeing the value in an intermediate, inconsistent state, with just part of the delta applied. You may choose to resolve this issue by having your application code synchronize on reads and writes.
- GemFire loses any reference to the old value because the old value is transformed in place into the new value. Because of this, your `CacheListener` sees the same new value returned for `EntryEvent.getOldValue` and `EntryEvent.getNewValue`.
- Exceptions thrown from `fromDelta` may leave your cache in an inconsistent state. Without cloning, any interruption of the delta application could leave you with some of the fields in your cached object changed and others unchanged. If you do not use cloning, keep this in mind when you program your error handling in your `fromDelta` implementation.

Example 13.1 Enabling Cloning in cache.xml

```
<region name="exampleRegion">
   <region-attributes refid="CACHING_PROXY" cloning-enabled="true"
      pool-name="examplePool"/>
</region>
```

Example 13.2 Enabling Cloning Through the C++ API

```
RegionFactoryPtr regionFactory =
   cachePtr->createRegionFactory(CACHING_PROXY);
RegionPtr regionPtr = regionFactory
   ->setCloningEnabled(true)
   ->create("myRegion");
```

13.5 Implementing Delta Propagation

By default, delta propagation is enabled in your distributed system. When enabled, delta propagation is used for objects that implement the delta interface. See *Delta Propagation API* on page 258. You program the client-side methods to extract delta information for your entries and to apply received delta information.

1. Choose whether to enable cloning. Cloning is disabled by default. See cloning-enabled (page 259).
   1.1 If you enable cloning, consider providing your own implementation, to optimize performance.
   1.2 If you do not enable cloning, be sure to synchronize your delta code.
2. Study your object types and expected application behavior to determine which objects should use delta propagation. Delta propagation is not beneficial for all data and data modification scenarios. For guidance on your decision, see Performance Considerations and Limitations (page 262).
3. If you do not enable cloning, review all associated listener code for dependencies on the entry event old value. Without cloning, GemFire modifies the entry in place and so loses its reference to the old value. For delta events, the `EntryEvent` methods to retrieve the old and new values both return the new value.
4. For every class where you want delta propagation, implement the delta interface and update your methods to support delta propagation. Exactly how you do this is depends on your application and object needs, but these steps describe the basic approach:
   4.1 Study the object contents to decide how to handle delta changes. Delta propagation has the same issues of distributed concurrency control as the distribution of full objects, but on a more detailed level. Some parts of your objects may be able to change independent of one another while others may always need to change together. Send deltas large enough to keep your data logically consistent. If, for example, field A and field B depend on each other, then your delta distributions should either update both fields or neither. As with regular updates, the fewer producers you have on a data region, the lower your likelihood of concurrency issues.
   4.2 In the application code that puts entries, put the fully populated object into the local cache. Even though you are planning to send only deltas, errors on the receiving end could cause GemFire to request the full object, so you must provide it to the originating `put` method. Do this even in empty producers, with regions configured for no local data storage. This usually means doing a `get` on the entry, unless you are sure it does not already exist anywhere in the distributed region.
   4.3 Change each field's update method to record information about the update. The information must be sufficient for `toDelta` to encode the delta and any additional required delta information when it is invoked.
   4.4 Write `hasDelta` to report on whether a delta is available.
   4.5 When writing your serialization methods, `toDelta`, `fromDelta`, `toData`, `fromData`, be sure to exclude any fields used to manage delta state, which do not need to be sent.
   4.6 Write `toDelta` to create a byte stream with the changes to the object and any other information `fromDelta` will need to apply the changes. Before returning from `toDelta`, reset your delta state to indicate that there are no delta changes waiting to be sent.
   4.7 Write `fromDelta` to decode the byte stream that `toDelta` creates and update the object.
   4.8 Make sure you provide adequate synchronization to your object to maintain a consistent object state. If you do not use cloning, you will probably need to synchronize on reads and writes to avoid reading partially written updates from the cache.This might involve `toDelta`, `fromDelta`, and other object access and update methods. Additionally, your implementation should take into account the possibility of concurrent invocations of `fromDelta` and one or more of the object's update methods.

13.6 Errors in Delta Propagation

Errors in delta propagation fall into two categories based on how they are handled:

1. Problems applying the delta that can be remedied by having the originating member send the full value in place of the delta. Your `put` operation does not see errors or exceptions related to the failed delta propagation. The system automatically does a full value distribution to the receiver where the problem occurs. This type of error includes:
    - Unavailable entry value in the receiving cache, either because the entry is missing or its value is null. In both cases, there is nothing to apply the delta to and the full value must be sent. This is most likely to occur if you destroy or invalidate your entries locally, either through application calls or through configured actions like eviction or entry expiration.
    - `InvalidDeltaException` thrown by `fromDelta` method, programmed by you. This exception enables you to avoid applying deltas that would violate data consistency checks or other application requirements. Throwing this exception causes GemFire to send the full value.
    - Any error applying the delta in a client in server-to-client propagation. The client retrieves the full value from the server.
2. Problems creating or distributing the delta that cannot be fixed by distributing the full value. These problems are caused by errors or exceptions in `hasDelta` or `toDelta`. In these cases, your `put` operation fails with an exception.

13.7 Performance Considerations and Limitations

The main purpose of delta propagation is to reduce the overhead of sending large amounts of data over the network.

Generally, the larger your objects and the smaller the deltas, the greater the benefits of using delta propagation. Partitioned regions generally benefit more with higher redundancy levels.

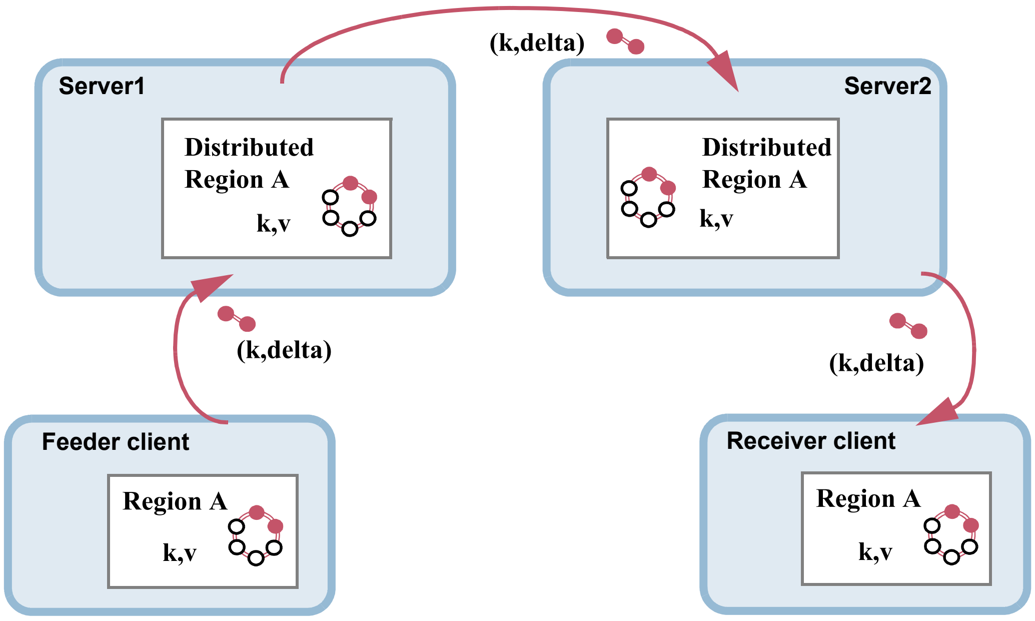
Delta propagation does not show any significant benefits in some application scenarios. On occasion it results in degradation. The main areas of consideration are:

- Added costs of deserializing your objects to apply deltas. Applying a delta requires the entry value to be deserialized. Once this is done, the object is stored back in the cache in deserialized form. This aspect of delta propagation only negatively impacts your system if your objects are not already being deserialized for other reasons, such as for indexing and querying or for listener operations. Once stored in deserialized form, there are reserialization costs for operations that send the object outside of the VM, like distribution across gateways, values sent in response to `netSearch` or client requests, and storage to disk. The more operations that require reserialization, the higher the overhead of deserializing the object.
- Cloning. Using cloning can affect performance. Not using cloning is risky, however, as you are modifying cached values in place. Make sure you synchronize your entry access to keep your cache from becoming inconsistent.
- Problems applying the delta that cause the system to go back to the originator for the full entry value. When this happens, the overall operation costs more than sending the full entry value in the first place. This can be additionally aggravated if your delta is sent to a number of recipients, all or most of them request a full value, and the full value send requires the object to be serialized.
- Disk I/O costs associated with overflow regions. If you use eviction with overflow to disk, on-disk values must be brought into memory in order to apply the delta. This is much more costly than removing the reference to the disk copy, as you would do with a full value distribution into the cache.

13.8 Examples of Delta Propagation

In this example, the feeder client is connected to the first server, and the receiver client is connected to the second. The servers are peers to each other.

Figure 13.2 Example of Delta Propagation in the Client/Server



These are the main operations shown in the example:

1. In the Feeder client, the application updates the entry object and puts the entry. In response to the `put`, GemFire calls `hasDelta`, which returns true, so GemFire calls `toDelta` and forwards the extracted delta to the server. If `hasDelta` returned false, GemFire would distribute the full entry value.
2. In Server1, GemFire applies the delta to the cache, distributes the received delta to the server's peers, and forwards it to any other clients with interest in the entry (there are no other clients to Server1 in this example)
3. In Server2, GemFire applies the delta to the cache and forwards it to its interested clients, which in this case is just Receiver client.

Client Example Files

These example files, from the product `quickstart` examples, show the basic approach to programming a delta propagation implementation for an entry value object, named `DeltaExample` in this example.

Example 13.3 XML file used for the examples

```
<cache>
   <region name="root" refid="CACHING_PROXY">
     <region-attributes cloning-enabled="true" pool-name="examplePool"/>
   </region>
   <pool name="examplePool" subscription-enabled="true" server-group=
"ServerGroup1">
       <locator host="localhost" port="34756"/>
   </pool>
</cache>
```

Example 13.4 Delta Example Implementation - C#

```
using System;
using GemStone.GemFire.Cache;
namespace GemStone.GemFire.Cache.QuickStart
{
   public class DeltaExample : IGFDelta, IGFSerializable, ICloneable
   {
      // data members
      private Int32 m_field1;
      private Int32 m_field2;
      private Int32 m_field3;

      // delta indicators
      private bool m_f1set;
      private bool m_f2set;
      private bool m_f3set;

      public DeltaExample(Int32 field1, Int32 field2, Int32 field3)
      {
         m_field1 = field1;
         m_field2 = field2;
         m_field3 = field3;
         reset();
      }
      public DeltaExample()
      {
         reset();
      }

      public DeltaExample(DeltaExample copy)
      {
         m_field1 = copy.m_field1;
         m_field2 = copy.m_field2;
         m_field3 = copy.m_field3;
         reset();
      }

      private void reset()
      {
```

```
                m_f1set = false;
                m_f2set = false;
                m_f3set = false;
            }

            public Int32 getField1()
            {
                return m_field1;
            }
    // REPEAT FOR OTHER FIELDS

            public void setField1(Int32 val)
            {
                lock(this)
                {
                    m_field1 = val;
                    m_f1set = true;
                }
            }
    // REPEAT FOR OTHER FIELDS

            public bool HasDelta()
            {
                return m_f1set || m_f2set || m_f3set;
            }
            public void ToDelta(DataOutput DataOut)
            {
                lock(this)
                {
                    DataOut.WriteBoolean(m_f1set);
                    if (m_f1set)
                    {
                        DataOut.WriteInt32(m_field1);
                    }
    // REPEAT FOR OTHER FIELDS

                    reset();
                }
            }
            public void FromDelta(DataInput DataIn)
            {
                lock(this)
                {
                    m_f1set = DataIn.ReadBoolean();
                    if (m_f1set)
                    {
                        m_field1 = DataIn.ReadInt32();
                    }
    // REPEAT FOR OTHER FIELDS

                }
            }
            public void ToData(DataOutput DataOut)
            {
                DataOut.WriteInt32(m_field1);
                DataOut.WriteInt32(m_field2);
                DataOut.WriteInt32(m_field3);
            }
            public IGFSerializable FromData(DataInput DataIn)
            {
               m_field1 = DataIn.ReadInt32();
               m_field2 = DataIn.ReadInt32();
               m_field3 = DataIn.ReadInt32();
               return this;
            }
            public UInt32 ClassId
            {
               get
               {
                  return 0x02;
               }
            }
            public UInt32 ObjectSize
            {
               get
               {
                  UInt32 objectSize = 0;
                  return objectSize;
               }
            }

            public static IGFSerializable create()
            {
               return new DeltaExample();
            }
            public Object Clone()
            {
               return new DeltaExample(this);
            }
         }
      }
```

Example 13.5 Delta Example Implementation - C++

```
#ifndef __Delta_Example__
#define __Delta_Example__

#include <gfcpp/GemfireCppCache.hpp>

using namespace gemfire;

class DeltaExample: public Cacheable, public Delta
{

private:

    // data members
    int32_t m_field1;
    int32_t m_field2;
    int32_t m_field3;

    // delta indicators
    mutable bool m_f1set;
    mutable bool m_f2set;
    mutable bool m_f3set;

public:

    DeltaExample(int32_t field1, int32_t field2, int32_t field3) :
       m_field1(field1), m_field2(field2), m_field3(field3)
    {
       reset();
    }

    DeltaExample()
    {
       reset();
    }

    DeltaExample(DeltaExample * copy)
    {
       m_field1 = copy->m_field1;
       m_field2 = copy->m_field2;
       m_field3 = copy->m_field3;
       reset();
    }

    void reset() const
    {
       m_f1set = false;
       m_f2set = false;
       m_f3set = false;
    }

    int getField1()
    {
       return m_field1;
    }
// REPEAT FOR OTHER FIELDS

    void setField1(int val)
```

```
            {
               lock();
               m_field1 = val;
               m_f1set = true;
               unlock();
            }
         // REPEAT FOR OTHER FIELDS

            virtual bool hasDelta()
            {
               return m_f1set || m_f2set || m_f3set;
            }

            virtual void toDelta(DataOutput& out) const
            {
               lock();

               out.writeBoolean(m_f1set);
               if (m_f1set)
               {
                  out.writeInt(m_field1);
               }
         // REPEAT FOR OTHER FIELDS

               reset();

               unlock();
            }

            virtual void fromDelta(DataInput& in)
            {
               lock();

               in.readBoolean(&m_f1set);
               if (m_f1set)
               {
                  in.readInt(&m_field1);
               }
         // REPEAT FOR OTHER FIELDS

               reset();

               unlock();
            }

            virtual void toData(DataOutput& output) const
            {
               lock();
               output.writeInt(m_field1);
               output.writeInt(m_field2);
               output.writeInt(m_field3);
               unlock();
            }

            virtual Serializable* fromData(DataInput& input)
            {
               lock();
               input.readInt(&m_field1);
               input.readInt(&m_field2);
               input.readInt(&m_field3);
               unlock();
               return this;
            }

            virtual int32_t classId() const
            {
               return 2;
            }

            virtual uint32_t objectSize() const
            {
               return 0;
            }

            DeltaPtr clone()
            {
               return DeltaPtr(new DeltaExample(this));
            }

            virtual ~DeltaExample()
            {
            }

            static Serializable* create()
            {
               return new DeltaExample();
            }

            void lock() const { /* add your platform dependent syncronization code
         here */ }
            void unlock() const { /* add your platform dependent syncronization code
         here */ }
         };
         #endif
```

Chapter

14 Programming Examples

This chapter provides a set of programming examples to help you understand the GemFire Enterprise native client API.

In this chapter:

14.1 Declaring a Native Client Region

The following example shows how to declare a native client region in a `cache.xml` file.

Example 14.1 Declaring a Native Client Region Using cache.xml

```
<cache>
   <region name = "root1" >
       <region-attributes refid="CACHING_PROXY" pool-name="poolName1"/>
   </region>
   <region name = "root2" >
       <region-attributes refid="PROXY" pool-name="poolName2"/>
   </region>
   <pool name="poolName1" subscription-enabled="true">
       <server host="localhost" port="40404" />
   </pool>
   <pool name="poolName2" subscription-enabled="true">
       <server host="localhost" port="40404" />
   </pool>
</cache>
```

- The pool defines a list of cache servers that the native client region can communicate with.
- The `CACHING_PROXY` setting causes the client region to cache data and to communicate with the servers. The PROXY setting causes the client region to communicate with the servers, but cache no data.
- The region `subscription-enabled` property, if `true`, indicates that the client should receive data updates when server data changes.
- Native clients do not specify cache loaders or writers, which are provided by the server.

14.2 API Programming Example — C#

This C# programming code in the next example demonstrates how to use two or more clients sharing a distributed region in a GemFire Enterprise cache.

Example 14.2 Demonstrating Gets and Puts Using the C# .NET API

```
using System;

// Use the GemFire namespace
using GemStone.GemFire.Cache;

namespace GemStone.GemFire.Cache.QuickStart
{
  // The BasicOperations QuickStart example.
  class BasicOperations
  {
    static void Main(string[] args)
    {
      try
      {
        // Create a GemFire Cache.
        CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(null);

        Cache cache = cacheFactory.SetSubscriptionEnabled(true).Create();

        Console.WriteLine("Created the GemFire Cache");

        RegionFactory regionFactory =
cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);

        Region region =  regionFactory.Create("exampleRegion");

        Console.WriteLine("Created the Region Programmatically.");

        // Put an Entry (Key and Value pair) into the Region using the
direct/shortcut method.
        region.Put("Key1", "Value1");

        Console.WriteLine("Put the first Entry into the Region");

        // Put an Entry into the Region by manually creating a Key and a
Value pair.
        CacheableInt32 key = new CacheableInt32(123);
        CacheableString value = new CacheableString("123");
        region.Put(key, value);

        Console.WriteLine("Put the second Entry into the Region");

        if (IntPtr.Size == 8) // Are we a 64 bit process?
        {
          Char ch = 'A';
          string text = new string(ch, 1024 * 1024);
          for (int item = 0; item < (5 * 1024 /* 5 GB */); item++)
          {
            region.LocalPut(item, text);
          }
          Console.WriteLine("Put over 4 GB data locally");
        }
// Get Entries back out of the Region.
        IGFSerializable result1 = region.Get("Key1");

        Console.WriteLine("Obtained the first Entry from the Region");

        IGFSerializable result2 = region.Get(key);

        Console.WriteLine("Obtained the second Entry from the Region");

        // Invalidate an Entry in the Region.
        region.Invalidate("Key1");

        Console.WriteLine("Invalidated the first Entry in the Region");

        // Destroy an Entry in the Region.
        region.Destroy(key);

        Console.WriteLine("Destroyed the second Entry in the Region");

        // Close the GemFire Cache.
        cache.Close();

        Console.WriteLine("Closed the GemFire Cache");
      }
      // An exception should not occur
      catch (GemFireException gfex)
      {
        Console.WriteLine("BasicOperations GemFire Exception: {0}",
gfex.Message);
      }
    }
  }
}
```

14.3 API Programming Example — C++

The next example uses the C++ API to implement a cache loader, which is generally used to retrieve data from an outside source.

Example 14.3 Implementing a Cache Loader Using the C++ API

```
CacheablePtr TestCacheLoader::load(
  const RegionPtr& region,
  const CacheableKeyPtr& key,
  const UserDataPtr& aCallbackArgument)
{
  m_bInvoked = true;
  printf( "CacheLoader.load : %s\n", printEvent(region, key,
        aCallbackArgument).c_str());
  CacheablePtr value = NULLPTR;
  try {
    value = region->get(key, aCallbackArgument);
  } catch(Exception& ex) {
      fprintf(stderr, "Exception in TestCacheCallback::printEvent [%s]\n",
ex.getMessage());
  }
  if (value != NULLPTR) {
     printf( "Loader found value: ");
     std::string formatValue = printEntryValue(value);
     printf( "%s\n",formatValue.c_str());
  } else {
     printf( " Loader did not find a value");
  }

  return value;
}
```

14.4 Data Serialization Examples

C++ Example

Example 14.4 Implementing an Embedded Object Using the C++ API

```
class User
  : public Serializable
{
private:
  std::string name;
  int32_t userId;
  ExampleObject *eo;
public:
  User( std::string name, int32_t userId )
    : name( name ),userId( userId )
  {
    eo = new ExampleObject(this->userId);
  }

  ~User() {
    if (eo != NULL) delete eo;
    eo = NULL;
  }

  User ()
  {
    name = "";
    userId = 0;
    eo = new ExampleObject(userId);
  }

  User( const char *strfmt, char delimeter )
  {
    std::string userId_str;
    std::string sValue(strfmt);
    std::string::size_type pos1 = sValue.find_first_of(delimeter);
    std::string::size_type pos2;
    if (pos1 == std::string::npos) {
      userId_str = sValue;
      name = sValue;
    }
    else {
      userId_str = sValue.substr(0, pos1);
      pos2 = sValue.find(delimeter, pos1+1);
      int len;
      if (pos2==std::string::npos) {
        len = sValue.length()-pos1;
      }
      else {
        len = pos2-pos1;
      }
      name = sValue.substr(pos1+1, len);
    }
    userId = (int32_t)atoi(userId_str.c_str());
    eo = new ExampleObject(userId_str);
  }
```

```
CacheableStringPtr toString() const
{
  CacheableStringPtr eo_str = eo->toString();
  char userId_str[128];
  sprintf(userId_str,"User: %d", userId);
  std::string sValue = std::string(userId_str) + "," + name + "\n";
  sValue += std::string(eo_str->asChar());
  return CacheableString::create( sValue.c_str() );
}

int32_t getUserId( )
{
  return userId;
}

std::string getName( )
{
  return name;
}

ExampleObject *getEO()
{
  return eo;
}

void setEO(ExampleObject *eObject)
{
  eo = eObject;
}

// Add the following for the Serializable interface
// Our TypeFactoryMethod

static Serializable* createInstance( )
{
  return new User(std::string("gester"), 123);
}

int32_t classId( ) const
{
  return 0x2d; // 45
}

void toData( DataOutput& output ) const
{
  output.writeASCII( name.c_str(), name.size() );
  output.writeInt( userId );
  eo->toData(output);
}

uint32_t objectSize( ) const
{
  return ( sizeof(char) * ( name.size() + 1 ) ) +
    sizeof(User) + eo->objectSize();
}

Serializable* fromData( DataInput& input )
{
        char *readbuf;
        input.readASCII( &readbuf );
        name = std::string(readbuf);
        input.freeUTFMemory( readbuf );
        input.readInt( &userId );
        eo->fromData(input);
        return this;
      }
    };
```

Example 14.5 Implementing Complex Data Types Using the C++ API

```
class ExampleObject
   : public Serializable
{
private:
  double double_field;
  float float_field;
  long long_field;
  int  int_field;
  short short_field;
  std::string string_field;
  std::vector<std::string> string_vector;
public:
  ExampleObject() {
    double_field = 0.0;
    float_field = 0.0;
    long_field = 0;
    int_field = 0;
    short_field = 0;
    string_field = "";
    string_vector.clear();
  }
  ~ExampleObject() {
  }
  ExampleObject(int id) {
    char buf[64];
    sprintf(buf, "%d", id);
    std::string sValue(buf);
    int_field = id;
    long_field = int_field;
    short_field = int_field;
    double_field = (double)int_field;
    float_field = (float)int_field;
    string_field = sValue;
    string_vector.clear();
    for (int i=0; i<3; i++) {
      string_vector.push_back(sValue);
    }
  }
  ExampleObject(std::string sValue) {
    int_field = atoi(sValue.c_str());
    long_field = int_field;
    short_field = int_field;
    double_field = (double)int_field;
    float_field = (float)int_field;
    string_field = sValue;
    string_vector.clear();
    for (int i=0; i<3; i++) {
      string_vector.push_back(sValue);
    }
  }
  CacheableStringPtr toString() const {
    char buf[1024];
    std::string sValue = "ExampleObject: ";
    sprintf(buf,"%f(double),%f(double),%ld(long),%d(int),%d(short),",
double_field,float_field,long_field,int_field,short_field);
    sValue += std::string(buf) + string_field + "(string),";
    if (string_vector.size() >0) {
      sValue += "[";
      for (unsigned int i=0; i<string_vector.size(); i++) {
        sValue += string_vector[i];
        if (i != string_vector.size()-1) {
          sValue += ",";
        }
      }
      sValue += "](string vector)";
    }
    return CacheableString::create( sValue.c_str() );
  }
  double getDouble_field() {
    return double_field;
  }
  float getFloat_field() {
    return float_field;
  }
  long getLong_field() {
    return long_field;
  }
  int getInt_field() {
    return int_field;
  }
  short getShort_field() {
    return short_field;
  }
  std::string & getString_field() {
    return string_field;
  }
  std::vector<std::string> & getString_vector( ) {
    return string_vector;
  }

  // Add the following for the Serializable interface
  // Our TypeFactoryMethod

  static Serializable* createInstance( ) {
    return new ExampleObject();
  }
  int32_t classId( ) const
  {
    return 0x2e; // 46
  }
  bool operator == ( const Serializable& other ) const {
    const ExampleObject& otherEO = static_cast<const ExampleObject&>( other
);
    return ( 0 == strcmp( otherEO.toString()->asChar(), toString()->asChar()
) );
  }
  uint32_t hashcode( ) const {
    return int_field;
  }

  uint32_t objectSize( ) const
  {
    uint32_t objectSize = sizeof(ExampleObject);
    objectSize += sizeof(char) * ( string_field.size() + 1 );
    size_t itemCount = string_vector.size();
    for( size_t idx = 0; idx < itemCount; idx++ ) {
      // copy each string to the serialization buffer, including the null
      // terminating character at the end of the string.
      objectSize += sizeof(char) * ( string_vector[idx].size() + 1 );
    }
    return objectSize;
  }

  void toData( DataOutput& output ) const {
    output.writeDouble( double_field );
    output.writeFloat( float_field );
    output.writeInt( (int64_t)long_field );
    output.writeInt( (int32_t)int_field );
    output.writeInt( (int16_t)short_field );
    output.writeASCII( string_field.c_str(), string_field.size());
    size_t itemCount = string_vector.size();
    output.writeInt( (int32_t) itemCount );
    for( size_t idx = 0; idx < itemCount; idx++ ) {
      // copy each string to the serialization buffer, including the null
      // terminating character at the end of the string.
      output.writeASCII( string_vector[idx].c_str(),
string_vector[idx].size() );
    }
  }

  Serializable* fromData( DataInput& input )
  {
    char *readbuf;
    input.readDouble( &double_field );
    input.readFloat( &float_field );
    input.readInt( (int64_t *)(void *)&long_field );
    input.readInt( (int32_t *)(void *)&int_field );
    input.readInt( (int16_t *)(void *)&short_field );

    int32_t itemCount = 0;
    input.readASCII( &readbuf );
    string_field = std::string(readbuf);
    input.freeUTFMemory( readbuf );

    string_vector.clear();
    input.readInt( (int32_t*) &itemCount );
    for( int32_t idx = 0; idx < itemCount; idx++ ) {
      // read from serialization buffer into a character array
      input.readASCII( &readbuf );
      // and store in the history list of strings.
      string_vector.push_back( readbuf );
      input.freeUTFMemory( readbuf );
    }
    return this;
  }
};
typedef SharedPtr<ExampleObject> ExampleObjectPtr;
```

C# Example

Example 14.6 Implementing a User-Defined Serializable Object Using the C# API

```
class User : IGFSerializable
  {
    private string m_name;
    private int m_userId;
    ExampleObject m_eo;

    public User(string name, int userId)
    {
      m_name = name;
      m_userId = userId;
      m_eo = new ExampleObject();
    }
    public User()
    {
      m_name = string.Empty;
      m_userId = 0;
      m_eo = new ExampleObject();
    }

    public int UserId
    {
      get
      {
        return m_userId;
      }

    }

    public string Name
    {
      get
      {
        return m_name;
      }
    }

    public ExampleObject EO
    {
      get
      {
        return m_eo;
      }
      set
      {
        m_eo = value;
      }
    }

    public override string ToString()
    {
      return string.Format("User: {0}, {1}\n{2}", m_userId, m_name,
        m_eo.ToString());
    }
```

```
        // Our TypeFactoryMethod
        public static IGFSerializable CreateInstance()
        {
          return new User();
        }

        #region IGFSerializable Members

        public UInt32 ClassId()
        {
           get
           {
              return 45; // must match Java
            }
         }
        public IGFSerializable FromData(DataInput input)
        {
          m_name = input.ReadUTF();
          m_userId = input.ReadInt32();
          m_eo.FromData(input);
          return this;
        }

        public void ToData(DataOutput output)
        {
          output.writeUTF(m_name);
          output.writeInt32(m_userId);
          eo.ToData(output);
        }

        #endregion
      }
```

Java Example

Example 14.7 Implementing an Embedded Object Using the Java API

```
public class User implements DataSerializable
{
   private String name;
   private int userId;
   private ExampleObject eo;
   static
   {
      Instantiator.register(
         new Instantiator(User.class, (byte)45)
      {
         public DataSerializable newInstance()
         {
            return new User();
         }
      }
   );
   }
   /**
* Creates an "empty" User whose contents are filled in by
* invoking its toData() method
*/
   private User()
      {
         this.name = "";
         this.userId = 0;
      this.eo = new ExampleObject(0);
   }
   public User(String name, int userId)
   {
      this.name = name;
      this.userId = userId;
      this.eo = new ExampleObject(userId);
   }
   public void setEO(ExampleObject eo)
   {
      this.eo = eo;
   }
   public ExampleObject getEO()
   {
      return eo;
   }
   public void toData(DataOutput out) throws IOException
   {
      out.writeUTF(this.name);
      out.writeInt(this.userId);
      eo.toData(out);
   }
   public void fromData(DataInput in) throws IOException,
   ClassNotFoundException
   {
      this.name = in.readUTF();
      this.userId = in.readInt();
      this.eo.fromData(in);
   }
   public int getUserId()
   {
      return userId;
   }
   public String getName()
   {
      return name;
   }
   public boolean equals(User o)
   {
      if (!this.name.equals(o.name)) return false;
      if (this.userId != o.userId) return false;
      if (!this.eo.equals(o.eo)) return false;
      return true;
   }
}
```

Example 14.8 Implementing Complex Data Types Using the Java API

```
public class ExampleObject implements DataSerializable {
   private double double_field;
   private long long_field;
   private float float_field;
   private int int_field;
   private short short_field;
   private java.lang.String string_field;
   private Vector string_vector;
   static {
      Instantiator.register(new Instantiator(ExampleObject.class, (byte) 46)
      {
         public DataSerializable newInstance() {
            return new ExampleObject();
         }
      });
   }
   public ExampleObject( ) {
   this.double_field = 0.0D;
   this.long_field = 0L;
   this.float_field = 0.0F;
   this.int_field = 0;
   this.short_field = 0;
   this.string_field = null;
   this.string_vector = null;
   }
   public ExampleObject(int id) {
   this.int_field = id;
   this.string_field = String.valueOf(id);
   this.short_field = Short.parseShort(string_field);
   this.double_field = Double.parseDouble(string_field);
   this.float_field = Float.parseFloat(string_field);
   this.long_field = Long.parseLong(string_field);
   this.string_vector = new Vector();
   for (int i=0; i<3; i++) {
   this.string_vector.addElement(string_field);
   }
   }
   public ExampleObject(String id_str) {
   this.int_field = Integer.parseInt(id_str);
   this.string_field = id_str;
   this.short_field = Short.parseShort(string_field);
   this.double_field = Double.parseDouble(string_field);
   this.float_field = Float.parseFloat(string_field);
   this.long_field = Long.parseLong(string_field);
   this.string_vector = new Vector();
   for (int i=0; i<3; i++) {
      this.string_vector.addElement(string_field);
   }
   }
   public double getDouble_field( ) {
   return this.double_field;
   }
   public void setDouble_field( double double_field ) {
   this.double_field = double_field;
   }
   public long getLong_field( ) {
   return this.long_field;
```

```
}
public void setLong_field( long long_field ) {
this.long_field = long_field;
}
public float getFloat_field( ) {
return this.float_field;
}
public void setFloat_field( float float_field ) {
this.float_field = float_field;
}
public int getInt_field( ) {
return this.int_field;
}
public void setInt_field( int int_field ) {
this.int_field = int_field;
}
public short getShort_field( ) {
return this.short_field;
}
public void setShort_field( short short_field ) {
this.short_field = short_field;
}
public java.lang.String getString_field( ) {
return this.string_field;
}
public void setString_field( java.lang.String string_field ) {
this.string_field = string_field;
}
public Vector getString_vector( ) {
return this.string_vector;
}
public void setString_vector( Vector string_vector ) {
this.string_vector = string_vector;
}
public void toData(DataOutput out) throws IOException
{
   out.writeDouble(double_field);
   out.writeFloat(float_field);
   out.writeLong(long_field);
   out.writeInt(int_field);
   out.writeShort(short_field);
   out.writeUTF(string_field);
   out.writeInt(string_vector.size());
   for (int i = 0; i < string_vector.size(); i++)
   {
      out.writeUTF((String)string_vector.elementAt(i));
   }
}
public void fromData(DataInput in) throws IOException,
ClassNotFoundException
{
   this.double_field = in.readDouble();
   this.float_field = in.readFloat();
   this.long_field = in.readLong();
   this.int_field = in.readInt();
   this.short_field = in.readShort();
   this.string_field = in.readUTF();
   this.string_vector = new Vector();
   int size = in.readInt();
```

```
            for (int i = 0; i < size; i++)
            {
               String s = in.readUTF();
               string_vector.add(i, s);
            }
         }
         public boolean equals(ExampleObject o)
         {
            if (this.double_field != o.double_field) return false;
            if (this.float_field != o.float_field) return false;
            if (this.long_field != o.long_field) return false;
            if (this.int_field != o.int_field) return false;
            if (this.short_field != o.short_field) return false;
            if (!this.string_field.equals(o.string_field)) return false;
            if (!this.string_vector.equals(o.string_vector)) return false;
            return true;
         }
      }
```

Appendix

A

Installing the Berkeley DB Persistence Manager

This appendix describes how to download, build and install the Berkeley database libraries for use with disk overflow. The commands that you type are shown in **`boldface fixed`** font. See *Berkeley DB Persistence Manager* on page 58 for additional information about the Berkeley database libraries.

In this appendix:

A.1 Linux Installation

The *productDir* directory refers to the path to the native client product directory.

The following libraries must be present in the runtime linking path:

- `libBDBImpl.so` is provided in *productDir*/`lib`, so it is already present in the runtime linking path.
- `libdb_cxx-4.4.so` is the Berkeley DB Library, C++ version 4.4.20. This library needs to be created by the user separately and made available in the runtime linking path, or copied to *productDir*/`lib`, as described below.

Downloading, Building and Installing the Library

The Berkeley DB library (version 4.4.20) is created by downloading and compiling the source code.

1. Download source code for Berkeley DB version 4.4.20 from http://www.oracle.com/technology/software/products/berkeley-db/db/index.html
2. Unzip and untar the source code. Follow the “Build, Installation, and Upgrading Guide” instructions at http://www.oracle.com/technology/documentation/berkeley-db/db/ref/build_unix/intro.html to build the library.

   2.1 Run the `configure` command for 32-bit or 64-bit with these options, all entered on a single line. Change the `--prefix` directory specification to the location where you want the libraries:

   32-bit:

   ```
   ../dist/configure --enable-cxx --prefix=/binary_location/bdb-binaries
   ```

   64-bit:

   ```
   ../dist/configure --enable-cxx --prefix=/binary_location/bdb-binaries
   CFLAGS="-m64"
   ```

   2.2 Run `make install` as described in the build instructions. The libraries will be available in the `bdb-binaries` directory that you specified.
3. Copy /*binary_location*/`bdb-binaries/lib/libdb_cxx-4.4.so` to *productDir*/`lib`.

A.2 Windows Installation

The *productDir* directory refers to the native client product directory path.

The following libraries are required. The *productDir*\bin directory containing these libraries must be present in the Windows PATH environment variable, and that directory is added to PATH during the GemFire product installation.

- The file bdbimpl.dll is provided in *productDir*\bin.
- libdb44.dll is the Berkeley DB Library version 4.4.20. This library needs to be obtained by the user separately, as described below. The library can be copied to *productDir*\bin.

Verify that your PATH variable points to *productDir*\bin, and set it if it doesn't.

```
set PATH=productDir\bin;%PATH%
```

Downloading and Installing the Library

The Berkeley DB library (version 4.4.20) for Windows can be obtained in binary form as follows:

1. Download the source code for Berkeley DB version 4.4.20 from:

   http://www.oracle.com/technology/software/products/berkeley-db/db/index.html
2. From the folder "db-4.4.20\build_win32" Choose /File -> Open -> Project/Solution.../. In the *build_win32*, select *Berkeley_DB* and click Open
3. You will be prompted to convert the project files to current Visual C++ format. Select "Yes to All".
4. For 64-bit only, choose the project configuration from the drop-down menu on the tool bar ("Debug AMD64", "Release AMD64"). Change the CPU type from Win32 to x64.
5. Right-click on build_all target and select Build.

A.3 Solaris Installation

The *productDir* directory refers to the path to the native client product directory.

The following libraries must be present in the runtime linking path:

- `libBDBImpl.so` is provided in *productDir*/`lib`, and so is already present in the runtime linking path.
- `libdb_cxx-4.4.so` is the Berkeley DB Library, C++ version 4.4.20. This library needs to be created by the user separately and made available in the runtime linking path, or copied to *productDir*/`lib`, as described below.

Downloading, Building and Installing the Library

The Berkeley DB library (version 4.4.20) is created by downloading and compiling the source code.

1. Download source code for Berkeley DB version 4.4.20 from http://www.oracle.com/technology/software/products/berkeley-db/db/index.html
2. Unzip and untar the source code. Follow the instructions at http://www.oracle.com/technology/documentation/berkeley-db/db/ref/build_unix/intro.html to build the library.

   2.1 Run the `configure` command for 32-bit or 64-bit with the following options, all entered on a single line:

   32-bit:

```
../dist/configure CC=cc CXX=CC CXXFLAGS="-xarch=v8plus -library=stlport4"
CFLAGS="-xarch=v8plus" LDFLAGS="-xarch=v8plus" CXXLDFLAGS="-xarch=v8plus
-library=stlport4" --enable-cxx
```

   64-bit:

```
../dist/configure CC=cc CXX=CC CXXFLAGS="-xarch=v9 -library=stlport4
"CFLAGS="-xarch=v9" LDFLAGS="-xarch=v9" CXXLDFLAGS="-xarch=v9 -library
stlport4" --enable-cxx
```

   2.2 Run `make install` as described in the build instructions. The libraries will be available in the directory /*binary_location*/`bdb-binaries`.

3. Copy the file /*binary_location*/`bdb-binaries/lib/libdb_cxx-4.4.so` to *productDir*/`lib`.

Appendix B

gfcpp.properties Example File

The `gfcpp.properties` file provides a way to configure distributed system connections for the GemFire Enterprise native client. The following example shows the format of a `gfcpp.properties` file. The first two attributes in this example should be set by programmers during application development, while other attributes are set on-site during system integration. The properties and their default settings that can be set in this file are described in detail in Table 6.2 on page 145.

Example B.1 gfcpp.properties File Format

```
#Tue Feb 14 17:24:02 PDT 2006
log-level=info
cache-xml-file=./cache.xml
stacktrace-enabled=true
```

B.1 Using the Default Sample File

A sample `gfcpp.properties` file is included with the GemFire Enterprise native client installation in the *`productDir`*`/defaultSystem` directory. Example B.2 on page 294 lists the contents of that file. To use this file:

1. Copy the file to the directory where you start the application.
2. Uncomment the lines you need and edit the settings as shown in this example:

   ```
   cache-xml-file=test.xml
   ```

3. Start the application.

Example B.2 Default gfcpp.properties File

```
# Default Gemfire C++ distributed system properties
# Copy to current directory and uncomment to override defaults.
#
## Debugging support, enables stacktraces in gemfire::Exception.
#
# The default is false, uncomment to enable stacktraces in exceptions.
#stacktrace-enabled=true
#crash-dump-enabled=true
#
## License selection
#
#license-file=gfcppLicense.zip
#license-type=evaluation
#
## Cache region configurtion
#
#cache-xml-file=cache.xml
#
## Log file config
#
#log-file=gemfire_cpp.log
#log-level=config
# zero indicates use no limit.
#log-file-size-limit=0
# zero indicates use no limit.
#log-disk-space-limit=0
#
## Statistics values
#
# the rate is in seconds.
#statistic-sample-rate=1
#statistic-sampling-enabled=true
#statistic-archive-file=statArchive.gfs
# zero indicates use no limit.
#archive-file-size-limit=0
# zero indicates use no limit.
#archive-disk-space-limit=0
#enable-time-statistics=false
#
## Heap based eviction configuration
#
# maximum amount of memory used by the cache for all regions, 0 disables
this feature
#heap-lru-limit=0
# percentage over heap-lru-limit when LRU will be called.
#heap-lru-delta=10
#
## Durable client support
#
#durable-client-id=
#durable-timeout=300
#
## SSL socket support
#
#ssl-enabled=false
#ssl-keystore=
#ssl-truststore=
#
## .NET AppDomain support
#
#appdomain-enabled=false
#
## Misc
#
#conflate-events=server
#disable-shuffling-of-endpoints=false
#grid-client=false
#max-fe-threads=
#max-socket-buffer-size=66560
# the units are in seconds.
#connect-timeout=59
#notify-ack-interval=10
#notify-dupcheck-life=300
#ping-interval=10
#redundancy-monitor-interval=10
#auto-ready-for-events=true
#
## module name of the initializer pointing to sample
## implementation from templates/security
#security-client-auth-library=securityImpl
## static method name of the library mentioned above
#security-client-auth-factory=createUserPasswordAuthInitInstance
## credential for Dummy Authenticator configured in server.
## note: security-password property will be inserted by the initializer
## mentioned in the above property.
#security-username=root
```

B.2 Search Path for Multiple gfcpp.properties Files

The native client and cache server processes first look for their properties file in the *`productDir`*/`defaultSystem` directory, then in the working directory. Any properties set in the working directory override settings in the `defaultSystem/gfcpp.properties` file.

If you are running multiple processes on one machine, you can configure the `gfcpp.properties` file in the `defaultSystem` directory as a shared file that all processes can find. If a few processes need a slightly different configuration, you can put individual `gfcpp.properties` files in their home directories to override specific properties.

B.3 Overriding gfcpp.properties Settings

Application developers have the option of configuring system attributes programmatically, rather than using the `gfcpp.properties` file. Attributes set programmatically override any matching attribute settings in the `gfcpp.properties` file, but additional attributes not set programmatically will be configured using the settings in `gfcpp.properties`. For more information, see Defining Properties Programmatically (page 149).

Appendix C

Query Language Grammar and Reserved Words

This appendix lists the reserved words and the grammar for the query language used in the GemFire Enterprise native client.

Reserved KeyWords

These words are reserved for the query language and may not be used as identifiers. The words with an asterisk (`*`) after them are not currently used by the native client, but are reserved for future implementations.

`abs*`	`desc*`	`intersect*`	`orelse*`
`all*`	`dictionary`	`interval*`	`query*`
`and`	`distinct`	`is_defined`	`select`
`andthen*`	`double`	`is_undefined`	`set`
`any*`	`element`	`last*`	`short`
`array`	`enum*`	`like*`	`some*`
`as`	`except*`	`list*`	`string`
`asc*`	`exists*`	`listtoset*`	`struct*`
`avg*`	`false`	`long`	`sum*`
`bag*`	`first*`	`map`	`time`
`boolean`	`flatten*`	`max*`	`timestamp`
`by*`	`float`	`min*`	`true`
`byte`	`for*`	`mod*`	`type`
`char`	`from*`	`nil`	`undefine*`
`collection`	`group*`	`not`	`undefined`
`count*`	`having*`	`null`	`union*`
`date`	`import`	`octet`	`unique*`
`declare*`	`in`	`or`	`where`
`define*`	`int`	`order*`	

Language Grammar

symbol ::= *expression*

n	A nonterminal symbol that has to appear at some place within the grammar on the left side of a rule. All nonterminal symbols have to be derived to be terminal symbols.
t	The terminal symbol **t** (shown in **bold**)
x y	x followed by y
x | y	x or y
(x | y)	x or y
[x]	x or empty
{ x }	possibly empty sequence of x

```
query_program ::= [ imports ; ]  query  [semicolon]
imports ::= import { ; import }
import ::= IMPORT qualifiedName [ AS identifier ]
query ::= selectExpr  | expr
selectExpr ::= SELECT DISTINCT projectionAttributes fromClause [ whereClause ]
projectionAttributes ::= * | projectionList
projectionList ::= projection { comma projection }
projection ::= field | expr [ AS identifier ]
field ::= identifier colon expr
fromClause ::= FROM iteratorDef { comma iteratorDef }
iteratorDef ::= expr [ [ AS ] identifier ] [ TYPE identifier ]
                    |  identifier IN expr  [ TYPE identifier ]
whereClause ::= WHERE expr
expr ::= castExpr
castExpr ::= orExpr | left_paren identifier right_paren castExpr
orExpr ::= andExpr { OR andExpr }
andExpr ::= equalityExpr { AND equalityExpr }
equalityExpr ::= relationalExpr { ( = | <>  | != ) relationalExpr }
relationalExpr ::= inExpr { ( < | <= | > | >= ) inExpr }
inExpr ::= unaryExpr { IN unaryExpr }
unaryExpr ::= [ NOT ] unaryExpr
postfixExpr ::=  primaryExpr { left_bracket expr right_bracket }
                    | primaryExpr { dot identifier [ argList ] }
argList ::= left_paren [ valueList ] right_paren
qualifiedName ::= identifier { dot identifier }
primaryExpr ::= functionExpr
                    | identifier [ argList ]
                    | undefinedExpr
                    | collectionConstruction
                    | queryParam
                    | literal
                    | ( query )
                    | region_path
conversionExpr ::= ELEMENT left_paren query right_paren
                    | NVL left_paren query comma query right_paren
                    | TO_DATE left_paren query right_paren
undefinedExpr ::= IS_UNDEFINED left_paren query right_paren
                    | IS_DEFINED left_paren query right_paren
collectionConstruction ::= SET left_paren [ valueList ] right_paren
valueList ::= expr { comma expr }
queryParam ::= $ integerLiteral
```

```
region_path ::= forward_slash region_name { forward_slash region_name }
region_name ::= name_character { name_character }
identifier ::= letter { name_character }
literal ::= booleanLiteral
                | integerLiteral
                | longLiteral
                | doubleLiteral
                | floatLiteral
                | charLiteral
                | stringLiteral
                | dateLiteral
                | timeLiteral
                | timestampLiteral
                | NULL
                | UNDEFINED
booleanLiteral ::= TRUE | FALSE
integerLiteral ::= [ dash ] digit { digit }
longLiteral ::= integerLiteral L
floatLiteral ::= [ dash ] digit { digit } dot digit { digit }
                         [ ( E | e ) [ plus | dash ] digit { digit } ] F
doubleLiteral ::= [ dash ] digit { digit } dot digit { digit }
                         [ ( E | e ) [ plus | dash ] digit { digit } ] [ D ]
charLiteral ::= CHAR single_quote character single_quote
stringLiteral ::= single_quote { character } single_quote
dateLiteral ::= DATE single_quote integerLiteral dash integerLiteral dash
                  integerLiteral single_quote
timeLiteral ::= TIME single_quote integerLiteral colon integerLiteral colon
                  integerLiteral single_quote
timestampLiteral ::= TIMESTAMP
                        single_quote integerLiteral dash integerLiteral dash
                        integerLiteral
                        integerLiteral colon integerLiteral colon digit {
                           digit }
                        [ dot digit { digit } ] single_quote
letter ::= any unicode letter
character ::= any unicode character except 0xFFFF
name_character ::= letter | digit | underscore
digit ::= any unicode digit
```

The following expressions are all terminal characters.

```
dot ::= .
left_paren ::= (
left_bracket ::= [
right_bracket ::= ]
right_bracket ::= ]
single_quote ::= ’
underscore ::= _
forward_slash ::= /
comma ::= ,
semicolon ::= ;
colon ::= :
dash ::= -
plus ::= +
```

Language Notes

- Query language keywords such as `SELECT`, `NULL`, and `DATE` are case-insensitive. Identifiers such as attribute names, method names, and path expressions are case-sensitive.
- Comment lines begin with `--` (double dash).
- Comment blocks begin with `/*` and ended with `*/`.
- String literals are delimited by single quotes. Embedded single quotes are doubled. Examples:

  ```
  'Hello' value = Hello
  'He said, ''Hello''' value = He said, 'Hello'
  ```

- Character literals begin with the `CHAR` keyword followed by the character in single quotation marks. The single quotation mark character itself is represented as `CHAR''''` (with four single quotation marks).
- For the `TIMESTAMP` literal, there is a maximum of nine digits after the decimal point.

Appendix

D

System Statistics

This appendix provides information on the GemFire installation standard statistics for caching and distribution activities.

GemFire Native Client System Statistics

Statistics that end with "`time`*" are time-based statistics. For performance reasons, by default, the system does not collect these. To enable time-based statistics gathering, set the system property* `enable-time-statistics` (page 147).

In this appendix:

- Sampling Statistics (page 301)
- System Performance Statistics (page 302)
- Operating System Statistics (page 305)

Sampling Statistics

When applications and cache servers join a distributed system, they indicate whether to enable statistics sampling and whether to archive the statistics that are gathered. For more information about configuring statistics, see *Configuration Options* on page 141. The following are statistics related to the statistic sampler.

`sampleCount`	Total number of samples taken by this sampler.
`sampleTime`	Total amount of time spent taking samples.
`StatSampler`	Statistics on the statistic sampler.

System Performance Statistics

This section discusses the statistics of primary importance for system performance. Statistics are collected for each application or cache server that connects to a distributed system.

Region Statistics

These methods help you to get the statistics of a region. The primary statistics are:.

`creates`	The total number of cache creates for this region.
`puts`	The total number of cache puts for this region.
`putTime`	Total time spent doing put operations for this region.
`putAll`	The total number of cache putAlls for this region.
`putAllTime`	Total time spent doing putAll operations for this region.
`gets`	The total number of cache gets for this region.
`getTime`	Total time spent doing get operations for this region.
`getAll`	The total number of cache getAlls for this region.
`getAllTime`	Total time spent doing getAll operations for this region.
`hits`	The total number of cache hits for this region.
`misses`	The total number of cache misses for this region.
`entries`	The current number of cache entries for this region.
`destroys`	The total number of cache destroys for this region.
`clears`	The total number of cache clears for this region.
`overflows`	The total number of cache overflows for this region to disk.
`retrieves`	The total number of cache entries fetched from disk into the cache region.
`nonSingleHopCount`	The total number of times client request required multiple hops.
`metaDataRefreshCount`	The total number of times metadata was refreshed due to the observation of multiple hops.
`cacheLoaderCallsCompleted`	Total number of times a load has completed for this region.
`cacheLoaderCallTIme`	Total time spent invoking the loaders for this region.
`CacheWriterCallsCompleted`	Total number of times a cache writer call has completed for this region.
`CacheWriterCallTime`	Total time spent doing cache writer calls.
`CacheListenerCallsCompleted`	Total number of times a cache listener call has completed for this region.
`CacheListenerCallTime`	Total time spent doing cache listener calls for this region.

Cache Performance Statistics

Statistics on the GemFire cache (available if the member creates a cache). These can be used to determine the type and number of cache operations being performed and how much time they consume. The primary statistics are:

`creates`	The total number of cache creates.
`puts`	The total number of cache puts.
`gets`	The total number of cache gets.
`entries`	The current number of cache entries.
`hits`	The total number of cache hits.
`misses`	The total number of cache misses.
`destroys`	The total number of cache destroys.
`overflows`	The total number of cache overflows to persistence backup.
`cacheListenerCallsCompleted`	Total number of times a cache listener call has completed.

Continuous Query Statistics

Using these statistics, you can get information about a registered Continuous Query (CQ) represented by the CqQuery object.:

`inserts`	The total number of inserts for this CQ query.
`updates`	The total number of updates for this CQ query.
`deletes`	The total number of deletes for this CQ query.
`events`	The total number of events for this CQ query.

CQ Service Statistics

Using these methods, you can get aggregate statistical information about the continuous queries of a client.:

`CqsActive`	The total number of CQs active this CQ query.
`CqsCreated`	The total number of CQs created for this CQ service.
`CqsClosed`	The total number of CQs closed for this CQ service.
`CqsStopped`	The total number of CQs stopped for this CQ service.
`CqsOnClient`	The total number of Cqs on the client for this CQ service.

Pool Statistics

The pool object provides the following statistics.

`locators`	Current number of locators discovered.
`servers`	Current number of servers discovered.
`servers`	Number of servers hosting this clients subscriptions.
`locatorRequests`	Number of requests from this connection pool to a locator.
`locatorResponses`	Number of responses from the locator to this connection pool.
`poolConnections`	Current number of pool connections.
`connects`	Total number of times a connection has been created.
`ConnectionWaitTime`	Total time (nanoseconds) spent waiting for a connection.
`disconnects`	Total number of times a connection has been destroyed.
`minPoolSizeConnects`	Total number of connects done to maintain minimum pool size.
`loadConditioningConnects`	Total number of connects done due to load conditioning.
`idleDisconnects`	Total number of disconnects done due to idle expiration.
`loadConditioningDisconnects`	Total number of disconnects done due to load conditioning expiration.
`connectionWaitsInProgress`	Current number of threads waiting for a connection.
`connectionWaits`	Total number of times a thread completed waiting for a connection (by timing out or by getting a connection).
`clientOpsInProgress`	Current number of clientOps being executed.
`clientOps`	Total number of clientOps completed successfully.
`clientOpFailures`	Total number of clientOp attempts that have failed.
`clientOpTimeouts`	Total number of clientOp attempts that have timed out.
`QueryExecutions`	Total number of queryExecutions.
`QueryExecutionTime`	Total time spent while processing queryExecution.

Delta Statistics

`deltaMessageFailures`	Total number of messages containing delta (received from server) but could not be processed after reception.
`deltaPuts`	Total number of puts containing delta that have been sent from client to server.
`processedDeltaMessages`	Total number of messages containing delta received from server and processed after reception.
`processedDeltaMessagesTime`	Total time spent applying delta (received from server) on existing values at client.

Operating System Statistics

The following sections provide operating system statistics on the VM's process. These can be used to determine the member's CPU, memory, and disk usage.

Linux Process Statistics

Using the following statistics, you can get information about a Linux operating system process that is using a GemFire system.

`imageSize`	The size of the process's image in megabytes.
`rssSize`	The size of the process's resident set size in megabytes.
`userTime`	The operating system statistic for the process CPU usage in user time
`systemTime`	The operating system statistic for the process CPU usage in system time.
`hostCpuUsage`	The operating system statistic for the host CPU usage.
`threads`	Number of threads currently active in this process.
`LinuxProcessStats`	Statistics for a Linux process.

Solaris Process Statistics

Using the following statistics, you can get information about a Solaris operating system process that is using a GemFire system.

`imageSize`	The size of the process's image in megabytes.
`rssSize`	The size of the process's resident set size in megabytes.
`userTime`	The operating system statistic for the process CPU usage in user time
`systemTime`	The operating system statistic for the process CPU usage in system time.
`processCpuUsage`	The operating system statistic for the CPU usage of this process.
`hostCpuUsage`	The operating system statistic for the host CPU usage.
`threads`	Number of threads currently active in this process.

Windows Process Statistics

Using the following statistics, you can get information about a Windows operating system process that is using a GemFire system.

`handles`	The total number of handles currently open by this process. This number is the sum of the handles currently open by each thread in this process.
`priorityBase`	The current base priority of the process. Threads within a process can raise and lower their own base priority relative to the process's base priority.
`threads`	Number of threads currently active in this process. An instruction is the basic unit of execution in a processor, and a thread is the object that executes instructions. Every running process has at least one thread.

`activeTime`	The elapsed time in milliseconds that all of the threads of this process used the processor to execute instructions. An instruction is the basic unit of execution in a computer, a thread is the object that executes instructions, and a process is the object created when a program is run. Code executed to handle some hardware interrupts and trap conditions are included in this count.
`pageFaults`	The total number of Page Faults by the threads executing in this process. A page fault occurs when a thread refers to a virtual memory page that is not in its working set in main memory. This will not cause the page to be fetched from disk if it is on the standby list and hence already in main memory, or if it is in use by another process with whom the page is shared.
`pageFileSize`	The current number of bytes this process has used in the paging file(s). Paging files are used to store pages of memory used by the process that are not contained in other files. Paging files are shared by all processes, and lack of space in paging files can prevent other processes from allocating memory.
`pageFileSizePeak`	The maximum number of bytes this process has used in the paging file(s). Paging files are used to store pages of memory used by the process that are not contained in other files. Paging files are shared by all processes, and lack of space in paging files can prevent other processes from allocating memory.
`privateSize`	The current number of bytes this process has allocated that cannot be shared with other processes.
`systemTime`	The elapsed time in milliseconds that the threads of the process have spent executing code in privileged mode. When a Windows system service is called, the service will often run in Privileged Mode to gain access to system-private data. Such data is protected from access by threads executing in user mode. Calls to the system can be explicit or implicit, such as page faults or interrupts. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. These subsystem processes provide additional protection. Therefore, some work done by Windows on behalf of your application might appear in other subsystem processes in addition to the privileged time in your process.
`userTime`	The elapsed time in milliseconds that this process's threads have spent executing code in user mode. Applications, environment subsystems, and integral subsystems execute in user mode. Code executing in User Mode cannot damage the integrity of the Windows Executive, Kernel, and device drivers. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. These subsystem processes provide additional protection. Therefore, some work done by Windows on behalf of your application might appear in other subsystem processes in addition to the privileged time in your process.
`virtualSize`	Virtual Bytes is the current size in bytes of the virtual address space the process is using. Use of virtual address space does not necessarily imply corresponding use of either disk or main memory pages. Virtual space is finite, and by using too much, the process can limit its ability to load libraries.

`activeTime`	The elapsed time in milliseconds that all of the threads of this process used the processor to execute instructions. An instruction is the basic unit of execution in a computer, a thread is the object that executes instructions, and a process is the object created when a program is run. Code executed to handle some hardware interrupts and trap conditions are included in this count.
`pageFaults`	The total number of Page Faults by the threads executing in this process. A page fault occurs when a thread refers to a virtual memory page that is not in its working set in main memory. This will not cause the page to be fetched from disk if it is on the standby list and hence already in main memory, or if it is in use by another process with whom the page is shared.
`pageFileSize`	The current number of bytes this process has used in the paging file(s). Paging files are used to store pages of memory used by the process that are not contained in other files. Paging files are shared by all processes, and lack of space in paging files can prevent other processes from allocating memory.
`pageFileSizePeak`	The maximum number of bytes this process has used in the paging file(s). Paging files are used to store pages of memory used by the process that are not contained in other files. Paging files are shared by all processes, and lack of space in paging files can prevent other processes from allocating memory.
`privateSize`	The current number of bytes this process has allocated that cannot be shared with other processes.
`systemTime`	The elapsed time in milliseconds that the threads of the process have spent executing code in privileged mode. When a Windows system service is called, the service will often run in Privileged Mode to gain access to system-private data. Such data is protected from access by threads executing in user mode. Calls to the system can be explicit or implicit, such as page faults or interrupts. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. These subsystem processes provide additional protection. Therefore, some work done by Windows on behalf of your application might appear in other subsystem processes in addition to the privileged time in your process.
`userTime`	The elapsed time in milliseconds that this process's threads have spent executing code in user mode. Applications, environment subsystems, and integral subsystems execute in user mode. Code executing in User Mode cannot damage the integrity of the Windows Executive, Kernel, and device drivers. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. These subsystem processes provide additional protection. Therefore, some work done by Windows on behalf of your application might appear in other subsystem processes in addition to the privileged time in your process.
`virtualSize`	Virtual Bytes is the current size in bytes of the virtual address space the process is using. Use of virtual address space does not necessarily imply corresponding use of either disk or main memory pages. Virtual space is finite, and by using too much, the process can limit its ability to load libraries.

`virtualSizePeak`	The maximum number of bytes of virtual address space the process has used at any one time. Use of virtual address space does not necessarily imply corresponding use of either disk or main memory pages. Virtual space is however finite, and by using too much, the process might limit its ability to load libraries.
`workingSetSize`	The current number of bytes in the Working Set of this process. The Working Set is the set of memory pages touched recently by the threads in the process. If free memory in the computer is above a threshold, pages are left in the Working Set of a process even if they are not in use. When free memory falls below a threshold, pages are trimmed from Working Sets. If they are needed they will then be soft-faulted back into the Working Set before they are paged out to disk.
`workingSetSizePeak`	The maximum number of bytes in the Working Set of this process at any point in time. The Working Set is the set of memory pages touched recently by the threads in the process. If free memory in the computer is above a threshold, pages are left in the Working Set of a process even if they are not in use. When free memory falls below a threshold, pages are trimmed from Working Sets. If they are needed they will then be soft faulted back into the Working Set before they leave main memory.
`cpuUsage`	Percentage CPU used by this process.
`WindowsProcessStats`	Statistics for a Microsoft Windows process.

Glossary

API	**A**pplication **P**rogramming **I**nterface. GemFire provides APIs to cached data for C++ and .NET applications.
application program	A program designed to perform a specific function directly for the user or, in some cases, for another application program. GemFire applications use the GemFire application programming interfaces (APIs) to modify cached data.
cache	A cache created by an application or cache server process. For the process, its cache is the point of access to all caching features and the only view of the cache that is available. Cache creation requires membership in the distributed system. See also **local cache** and **remote cache**.
cache configuration file	An XML file that declares the initial configuration of a cache, commonly named `cache.xml`. C++ and .NET applications can configure the cache additionally through the GemFire programming APIs.
cache listener	User-implemented plug-in for receiving and handling region entry events. A region's cache listener is called after an entry in the local cache is modified.
cache loader	User-implemented plug-in for loading data into a region. A region's cache loader is used to load data that is requested of the region but is not available in the distributed system. For a distributed region, the loader that is used can be in a different cache from the one where the data-request operation originated. See also **cache writer** and **netSearch**.
cache server	A long-running, configurable caching process, generally used to serve cached data to the applications. Usually, cache servers are configured to operate as servers in a client-server typology and their regions are configured to be replicates. See also **server**.
cache writer	User-implemented plug-in intended for synchronizing the cache with an outside data source. A region's cache writer is a synchronous listener to cache data events. The cache writer has the ability to abort a data modification. See also **cache loader**.
caching enabled	Specifies whether data is cached in the region. GemFire gives you the option of running applications without entry caching. For example, you can configure a distributed system as a simple messaging service.
client	In a client-server topology, clients can connect to cache servers, create new regions on the cache server, and store data in the cache server region. Clients can also connect to existing

	regions on a cache server and do directed gets and puts on the cache server. Clients do not track membership information about other clients, nor do they share information with other clients.
concurrency level	An estimate of the number of threads expected to concurrently modify values in the region. The actual concurrency may vary; this value is used to optimize the allocation of system resources.
connection	What an application uses to access a GemFire distributed system. An application can connect to a GemFire system by calling the `DistributedSystem::connect` function with the appropriate parameter settings. An application must connect to a distributed system to gain access to GemFire functionality.
destroy	Remove an entry from a region or remove a region from a cache.
disk policy	Determines whether LRU entries exceeding the entries limit for a caching region are destroyed or written to disk.
distributed scope	Enables a region to automatically send entry value updates to remote caches and incorporate updates received from remote caches. The scope identifies whether distribution operations must wait for acknowledgement from other caches before continuing. A distributed region's **cache loader** and **cache writer** (defined in the local cache) can be invoked for operations originating in remote caches.
distributed system	One or more GemFire system members that have been configured to communicate with each other, forming a single, logical system. Also used for the object that is instantiated to create the connection between the distributed system members.
DTD	**D**ocument **T**ype **D**efinition. A language that describes the contents of a Standard Generalized Markup Language (SGML) document. The DTD is also used with XML. The DTD definitions can be embedded within an XML document or in a separate file.
entry	A data object in a region. A region entry consists of a key and a value. The value is either null (invalid) or an object. A region entry knows what region it is in. See also **region data**, **entry key**, and **entry value**.
entry key	The unique identifier for an entry in a region.
entry value	The data contained in an entry.
expiration	A cached object expires when its time-to-live or idle timeout counters are exhausted. A region has one set of expiration attributes for itself and one set for all region entries.
expiration action	The action to be taken when a cached object expires. The expiration action specifies whether the object is to be invalidated or destroyed, and whether the action is to be performed only in the local cache or throughout the distributed system. A destroyed object is completely removed from the cache. A region is invalidated by invalidating all entries contained in the region. An entry is invalidated by having its value marked as invalid. Expiration attributes are set at the region level for the region and at the entry level for entries. See also **idle timeout** and **time-to-live**.
factory method	An interface for creating an object which at creation time can let its subclasses decide which class to instantiate. The factory method helps instantiate the appropriate subclass by creating the correct object from a group of related classes.

idle timeout	The amount of time a region or region entry may remain in the cache unaccessed before being expired. Access to an entry includes any `get` operation and any operation that resets the entry's time-to-live counter. Region access includes any operation that resets an entry idle timeout, and any operation that resets the region's time-to-live. Idle timeout attributes are set at the region level for the region and at the entry level for entries. See also **time-to-live** and **expiration action**.
interest list	A mechanism that allows a region to maintain information about receivers for a particular key-value pair in the region, and send out updates only to those nodes. Interest lists are particularly useful when you expect a large number of updates on a key as part of the entry life cycle.
invalid	The state of an object when the cache holding it does not have the current value of the object.
invalidate	Remove only the value of an entry in a cache, not the entry itself.
listener	An event handler. The listener registers its interest in one or more events and is notified when the events occur.
load factor	A region attribute that sets initial parameters on the underlying hashmap used for storing region entries.
local cache	The part of the distributed cache that is resident in the current process. This term is used to differentiate the cache where a specific operation is being performed from other caches in the distributed system. See also **remote cache**.
local scope	Enables a region to hold a private data set that is not visible to other caches. See also **scope**.
LRU	**L**east **R**ecently **U**sed. Refers to a region entry or entries most eligible for eviction due to lack of interest by client applications.
LRU entries limit	A region attribute that sets the maximum number of entries to hold in a caching region. When the capacity of the caching region is exceeded, LRU is used to evict entries.
membership	Applications and cache servers connect to a GemFire distributed system by invoking the static function `DistributedSystem::connect`. Through this connection, the application gains access to the APIs for distributed data caches. When a C++ or .NET application connects to a distributed system, it specifies the system it is connecting to by indicating the communication protocol and address to use to find other system members.
netSearch	The method used by GemFire to search remote caches for a data entry that is not found in the local cache region. This operates only on distributed regions.
overflows	An eviction option that causes the values of LRU entries to be moved to disk when the region reaches capacity. See **disk policy**.
persistence manager	The persistence manager manages the memory-to-disk and disk-to-memory actions for LRU entries. See **overflows**.
region	A logical grouping of data within a cache. Regions are used to store data entries (see **entry**). Each region has a set of attributes governing activities such as expiration, distribution, data loading, events, and eviction control.

region attributes	The class of attributes governing the creation, location, distribution, and management of a region and its entries.
region data	All of the entries directly contained in the region.
region entry	See **entry**.
remote cache	Any part of the distributed cache that is resident in a process other than the current one. If an application or cache server does not have a data entry in the region in its local cache, it can do a `netSearch` in an attempt to retrieve the entry from the region in a remote cache. See also **local cache**.
scope	Region attribute. Identifies whether a region keeps its entries private or automatically sends entry value updates to remote caches and incorporates updates received from remote caches. The scope also identifies whether distribution operations must wait for acknowledgement from other caches before continuing. See also **distributed scope** and **local scope**.
serialization	The process of converting an object graph to a stream of bytes.
server	In a client-server topology, the server manages membership and allows remote operations. The server maintains membership information for its clients in the distributed system, along with information about peer applications and other servers in the system. See also **cache server**.
system member	A process that has established a connection to a distributed system.
time-to-live	The amount of time a region or region entry may remain in the cache without being modified before being expired. Entry modification includes creation, update, and removal. Region modification includes creation, update, or removal of the region or any of its entries. Time-to-live attributes are set at the region level for the region, and at the entry level for entries. See also **idle timeout** and **expiration action**.
XML	E**X**tensible **M**arkup **L**anguage. An open standard for describing data from the W3C, XML is a markup language similar to HTML. Both are designed to describe and transform data, but where HTML uses predefined tags, XML allows tags to be defined inside the XML document itself. Using XML, virtually any data item can be identified. The XML programmer creates and implements data-appropriate tags whose syntax is defined in a DTD file.

Index