Cincom

Cincom Smalltalk™

VisualWorks®

Opentalk Communication Layer Developer's Guide

VisualWorks 8.1

P46-0135-09

SIMPLIFICATION THROUGH INNOVATION

Part Number: P46-0135-09

Software Release 8.1

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About This Book

The *Opentalk Communication Layer Developer's Guide* provides the experienced VisualWorks® developer with information necessary to implement communications protocols, and to develop distributed applications using the Opentalk Communication Layer. The communication layer is also known as the Opentalk Base, and is a part of the full Opentalk suite of features.

Audience

This book is written for experienced Smalltalk developers who are exploring Opentalk for the first time. Readers should have a good understanding of VisualWorks. Please refer to the several VisualWorks manuals as needed. This document does not presume deep and extensive understanding of communication protocols, distributed systems, or the CORBA architecture, but background in these areas is extremely helpful, and several reference texts addressing them are suggested in Annotated References.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).

Example	Description
filename.xwd	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > New	Indicates the name of an item (**New**) on a menu (**File**).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help > About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help > About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **Copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to:

helpna@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Personal Use License (PUL) is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides an important public resource for VisualWorks developers:

- A mailing list for users of VisualWorks PUL, which serves a growing community of users. To subscribe or unsubscribe, send a message to: vwnc-request@cs.uiuc.edu.

  with the SUBJECT of **subscribe** or **unsubscribe**. You can then address emails to: vwnc@cs.uiuc.edu.

The Smalltalk news group, comp.lang.smalltalk, carries on general discussions about different dialects of Smalltalk, including VisualWorks, and is a good source for advice.

Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks:

http://www.cincomsmalltalk.com/main/products/visualworks/visualworks-tutorials/

The guide you are reading is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/main/products/documentation/

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

1

Opentalk Communication Layer

Opentalk is a VisualWorks add-on that provides a rich and extensible environment for the development, deployment, maintenance, and monitoring of distributed applications. Opentalk contains frameworks and components for creating and extending communication protocols, object services, remotely targeted user interfaces, remote development tools, and other architectural components common to distributed systems.

The Opentalk Communication Layer consists of those components that define the base communication framework, several Smalltalk-to-Smalltalk communication protocols, and a select set of base services. It is that part of Opentalk targeted, first, toward the needs of protocol developers, and second, toward the needs of distributed service, component, and application developers planning to build directly off the communication layer.

This chapter describes the requirements for the Opentalk Communication Layer, its installation, and the contents of its parcels.

Installation

The parcels of the Opentalk Communication Layer are contained in the $(VISALWORKS)\opentalk directory.

To load communication layer support it in its entirety, load the following parcels in the order shown:

1 Opentalk-Prerequisites

2 Opentalk-Core-Support

3 Opentalk-Core

4 Opentalk-STST

5 Opentalk-Groups

6 Opentalk-Core-Services

The first three parcels are the minimal required support for Opentalk, and are generally sufficient if you intend to develop and use your own protocols in the Opentalk framework. They are loaded as prerequisites of the subsequent parcels.

Additional parcels provide further extensions to the Opentalk environment.

Parcels Contents

Opentalk has a layered architecture. The Opentalk Communication Layer is foundational with respect to the other layers of Opentalk. It consists of:

- frameworks and components used to implement communication protocols,
- concrete implementations of protocols for Smalltalk-to-Smalltalk unicast, multicast and broadcast, and
- base services immediately required to make those protocols usable, namely, simple unicast and multicast event multiplexing services and a lightweight naming service.

The Opentalk Communication Layer is packaged as the following parcels in the VisualWorks distribution.

Opentalk-Prerequisites

The parcel Opentalk-Prerequisites defines the Opentalk namespace and implements methods in class Object that must be loaded early.

Opentalk-Core-Support

Opentalk-Core-Support contains base system extensions required by Opentalk. These extensions mostly fall into three broad categories:

- methods that enforce pass-by-name for selected tool classes;
- methods that establish default pass modes at major nodes in the VisualWorks class hierarchy;
- overrides of the default marshaling methods in those classes that take explicit advantage of double-dispatching in marshaling.

Opentalk-Core

Opentalk-Core defines the core abstract classes of the Opentalk communication framework. These are the classes that a protocol developer will usually extend to create new communication protocols. Included are the classes that define proxies and object references, object tables, message headers and message bodies, pass mode control wrappers, marshalers, transports, object adaptors, request brokers, broker configurations, and several Opentalk-specific exceptions.

Opentalk-STST

Opentalk-STST contains the classes required to implement Smalltalk-to-Smalltalk unicast. Included are the classes that define the transports, the marshaler, and the message types specific to the Smalltalk-to-Smalltalk protocols: requests, close connection requests, replies, and error replies.

Opentalk-Groups

Opentalk-Groups contains the extensions that implement Smalltalk-to-Smalltalk multicast and broadcast. It includes classes defining broadcast and multicast transports and configurations, the required additional message types, and the multicast event service.

Opentalk-Core-Services

Opentalk-Core-Services defines an abstract Opentalk service class, a naming service, exceptions specific the naming service, and a unicast event service.

Opentalk-Scheduling

This package defines a scheduler and scheduler policies for use alongside request brokers when user applications elect to take explicit control of worker process scheduling.

Opentalk Tools

Two lightweight tools are included in Opentalk:

- The OpentalkConsole supports the configuration, creation, and registration of all release-quality request brokers. Load parcel Opentalk-Tools-Console to access this tool.
- The OpentalkMonitor supports inspection of and registration for all the events generated by release-quality brokers and object adaptors. Load parcel Opentalk-Tools-Monitor to access this tool.

2

Basic Opentalk Concepts

This chapter is a brief overview of some of the basic concepts and communication patterns in distributed computing. Discussion of these concepts and patterns is intended to provide the background for and prolegomena to subsequent discussion of the critical classes in the Opentalk Communication Layer. Readers interested in more substantive discussion of distributed computing are urged to peruse several of the references listed in the annotated bibliography provided in Annotated References.

Distributed Systems

Software systems exist along a scale of complexity and integration. Distributed systems are at one end of that scale.

Irrespective of the degree of coupling displayed, both networked and distributed systems involve inter-process communication. Inter-process communication involves the transfer of data or code from the sending process to the receiving process. For transfer to occur, the two processes must share a communication channel and a communication protocol. The Opentalk Communication Layer provides components for building protocols as well as several already implemented ones.

Stand-alone Systems

Computers started out as stand-alone systems. In the study of stand-alone systems, one assumes a single processor, a single sequential process, a uniform memory access time, and a constant performance cost for every primitive function call. This is the model involved in the study of data structures, algorithms, and computational complexity.

Even if one assumes several processes or processors running on the same host, they are assumed to fail together and at once. The case is different in a communicating application running on multiple hosts, liable to a variety of failures in part.

Communicating systems

Communicating systems, in contrast, are composed of physically separated processes that communicate to achieve some end. The communication occurs through channels of restricted bandwidth and variable latency. Such systems are susceptible to both processor and channel failures. Together, these two failure locations produce a rich classification of partial and complete failure modes. There is also sharp distinction between the cost of local function calls and those calls involving communication costs.

The Opentalk Communication Layer is intended to support the development of communicating systems. Such systems can be usefully categorized into two broad types.

Networked Systems

The processes of networked systems communicate, but have only ephemeral knowledge of one another's existence, do not much care about one another's state, and cooperate on an intermittent basis. This is the relationship between a process running a network browser and another running a page server.

Networked systems are stand-alone systems that happen to communicate. They are like single people who date a lot. Designers interested in such systems are interested in defining, accessing, and improving useful, discrete, and available services or service components. They are interested in reliability at the component or service level.

There are two flavors of networked systems: client-server and peer-to-peer.

Client-Server Systems

In the typical client-server system, there is a designated server process. It exists at a fixed, well-known address. Other, client processes connect to the server and invoke the service it provides as needed. The server usually does not need to know the network addresses of the clients. Clients and servers are loosely coupled, and the clients do not have the autonomy possessed by the server, since they depend upon it and not vice versa.

Peer-to-Peer Systems

Peer-to-peer systems are networked systems specifically designed to take advantage of the existence of multiple processes or hosts existing in an environment characterized by unstable connectivity and unpredictable IP addresses. These systems consist of loosely coupled, autonomous processes. Peer-to-peer systems usually take advantage of redundancy—the existence of several hosts playing the same computational role—to address the reliability and availability of the services provided. A classic example is the Usenet.

Current peer-to-peer systems often employ custom communication protocols and possess special components to manage fluctuating host presence and are often targeted at utilizing the 'wasted' compute cycles of networked hosts. Some of these systems, those attempting to support communities that span organizational and national boundaries, employ cryptographic components, at several levels, to ensure anonymity.

Distributed Systems

The processes in properly distributed systems are, unlike those in networked systems, tightly coupled. The communicating processes care about whether the others are running or crashed, what state they are in, whether they are in agreement about critical state variables. The distinguishing mark of such systems is that they are actively engaged in maintaining the coherence of distributed state, and communicate frequently in order to maintain that coherence. An example is the relationship that exists between a primary and a backup service process in a simple fault tolerant system.

In these systems, synchronization and coordination problems loom large. They are like families with four active children, who play team sports that send them off to practices and out-of-town games, take separate private music lessons, attend four different schools, and don't drive yet. Developers interested in these kinds of systems are interested in integrating and harmonizing services and components, and in ensuring the consistent and reliable behavior of several, often replicated components, running on several geographically separated hosts.

Channels

The communicating processes reside on machines, conventionally called hosts. The communication channel that the processes must share—often called a circuit, a wire, a transmission line, or a link—is something that moves bits between machines.

Protocols

The communication protocol that the processes must share is, broadly, an agreement about how communication is to proceed. More particularly, each process sends and receives messages through a protocol stack or suite, and a protocol is a specification of the message sequences and the message formats logically used in communication at the same layer, between a sending layer and a receiving one. In fact, only the bottommost layer of a stack "communicates" and moves bits. Each level above the bottommost uses the services of the layer below, passing data and control to the lower layer as required, and all layers provide services to the layers above. A service in this context is nothing other than a well-defined API. A layer may provide several communication service APIs to layers above it.

In principle, it makes no difference at what layer in software or hardware a given layer in a protocol stack is implemented. For example, IIOP is a protocol layer on top of TCP/IP. In Distributed Smalltalk, IIOP is implemented in Smalltalk code. So too are the Smalltalk-to-Smalltalk communication protocols of Opentalk. But both of these protocol implementations rest upon and invoke TCP/IP and UDP protocol primitives implemented in the VisualWorks engine.

Transfer Protocols

Any useful protocol stack or suite contains a transport layer, the lowest layer at which the messages handled are logical units of information still meaningful to an application, rather than packets or cells. The latter are restricted-size or fixed-sized units of binary data, containing address information sufficient to identify the sending and the receiving host, and are meaningful at network or internet layers.

There are two common types of transfer protocols or services: connection-oriented and connectionless.

Each of these types of communication protocols or services can be implemented in terms of the other. For example, the NCS protocol implemented in VisualWorks Distributed Smalltalk is a connection-oriented protocol implemented on top of the connectionless UDP transport protocol.

Together, the two transport protocol types are sufficient to implement any communication pattern. Newer, high-performance protocol stacks, like Asynchronous Transfer Mode (ATM), typically provide an adaptation layer to transport protocols such as TCP and UDP. Thus, the framework provided by the Opentalk Communication Layer will be sufficient for several purposes, and for the foreseeable future.

Connection-Oriented Transport Protocols

Connection-oriented protocols, also called telephone protocols, establish a virtual connection between the sending and the receiving processes and use it to transmit a data stream. In these protocols the virtual connection must be established before any data is transmitted, and the connection must be closed when it is no longer needed. Hence the implementation of such protocols explicitly required connection management components. The TCP protocol in the Internet protocol suite is a connection-oriented protocol.

Connectionless Transport Protocols

Connectionless or mail or datagram protocols transmit messages called datagrams to specified destinations. They are usually less expensive than connection-oriented protocols, but also less reliable. If necessary, the protocol layer above the transport layer will detect missing or out-of-order datagrams and take corrective action. The UDP protocol of the Internet protocol suite is a connectionless protocol.

Synchronization

In addition to data or code transfer, inter-process communication may involve process synchronization. If the communication is synchronous or blocking, the sending process waits until the receiving process completes the activities entailed by message receipt and responds to the sender. If the communication is asynchronous or non-blocking, the sending process does not wait for a reply or acknowledgement. In either case the receiving process usually blocks when no incoming messages are present.

Patterns of Communication

Distributed systems can be understood and designed at the messaging level. This is the level at which distributed systems are discussed and analyzed in texts on distributed algorithms.

In that context, a distributed system is modeled as a set of nodes, arranged in a directed graph. Each node is associated with a process. Each process has states, a message generating function, and a state transition function. Each node in the graph has incoming and outgoing edges. Associated with each edge is a link that can contain at most one message. The state of any process is determined by the set of messages it has received and the order in which it has received them.

Though this abstract model is useful in the analysis of algorithms, several higher-level message patterns are so common in the construction of distributed systems that is far better to think in their terms. The most useful higher-level patterns are remote invocation, remote execution, and group multicast.

Remote Invocation

The remote invocation pattern is also known as the remote procedure call (RPC), and sometimes as the client-server communication model. Remote invocations usually have the same structure as ordinary function invocations: the caller relinquishes control to the invoked function at the time of the call and regains control when the function returns. A remote invocation is just a remote function invocation with the facade of a local one.

A RPC involves two messages:

- the transmission of a request from the client process to the remote server process, and
- the transmission of a reply from the server to the client.A request includes at least:
    - an identifier of the object to which the request is addressed,
    - the name of the function to be invoked, and
    - the arguments to that function.

Requests that expect replies must also include a request identifier.

A reply includes at least:

- the identifier of the request to which it is a response, and
- the return value, which may be either an object of the type expected, given normal execution, or an exception.

RPCs can be implemented over either connection-oriented protocols like TCP or connectionless ones like UDP. If implemented over UDP, the RPC layer must address the reliability of the communication.

Synchronous RPCs

RPCs are usually synchronous. The client process blocks until it receives a reply from the server.

Asynchronous RPCs

Asynchronous or non-blocking RPCs are useful in several cases. Since the client does not wait for a reply, the client and the server process can work in parallel. The performance benefits are further enhanced if the client can, over a short period, send several asynchronous RPC requests, one to each of several servers, which increases parallelism. If many client requests have the same destination, the client has the option of bundling several of them before transmission, to minimize the number of messages sent.

Asynchronous RPCs have two forms: without reply and with reply.

- Asynchronous RPCs without reply are typically used only for non-critical notifications.
- Asynchronous RPC with reply has a wider range of uses. If a client process is not going to block and yet receive a reply, some object must be defined to catch the reply when it arrives, and serve as a place-holder for it until it does. Such objects are called promises. Promises are implemented in VisualWorks by class Promise. Promises are either “blocked” or “ready,” and are created “blocked.” When the RPC returns, a Promise stores the returned value and becomes “ready.” The return value is stored immutably, so that it can be reclaimed more than once. If the client attempts the reclaim the promise while it is still “blocked,” the client process also blocks. Promises support the use of asynchronous RPCs with negligible impact on the structure of client code.

Remote Execution

Remote execution, or remote compilation and execution, are variants of remote invocation in which the data shipped to the server includes function definitions, as well as function names and function arguments. The function definitions may be expressed in a one of several formats. The formats used usually presuppose some degree of homogeneity: the communicating processes must be running code written in the same implementation language or upon the same platform type. The client defines the functions it wishes to have the server perform or store under some name for later invocation and execution. The server acts as an execution environment.

This approach can be very flexible, and it is comparatively easy to implement in languages like Smalltalk. It also has a long pedigree: PostScript employs this approach. It is partially echoed in the VisualWorks architecture: the image is a client of the engine, albeit a co-located one.

Group Multicast

In multicast, messages are sent to several target processes rather than just one. The target processes are said to belong to a multicast group. Even if there is no underlying hardware support for multicast, it can be implemented using sequential unicast. Group multicast is a useful structuring concept in distributed design however implemented. Note that multicast implementations usually make the decision, about whether the sender of the multicast will also receive it, on a relatively low level.

Multicast is useful for:

- locating object or services in a network,
- multiple updates, needed when several processes are interested in the same event, and
- achieving either fault tolerance or faster performance through the replication of services or state.

In a clean group multicast design, the objects in a group should not need to be aware of the distinction between object groups (the set of all the objects that belong to a group) and process groups (the set of all processes that contain objects belonging to the group).

Any powerful implementation of the concept of the multicast group will usually augment the basic multicast transport protocol to improve reliability (for example, best effort delivery rather than ‘send once’), to

enforce atomicity (all group members receive the message or none do), or to guarantee the delivery order of multicast messages. Causally ordered multicast is sufficient for most applications.

Multicast groups take several important forms.

In subscription groups, the member processes receive the same information from a multicast source. Group members do not reply to the source.

In peer groups, communication is directed from all objects in the group to all objects in the group. This is the standard pattern in cooperative work applications. Multicast loopback will be “on” in circumstances where the order of the updates is significant and message ordering is enforced by the multicast protocol in use.

Server groups are employed when client requests are multicast to all of the servers that can handle the client request. Usually, one server replies, using unicast. Two may reply if there is concern about rapid response in case of failure. There are several subvariations of these patterns, dependent on how the replying server set is elected. Server groups are one approach for ensuring that all servers have the same state. They require protocols enforcing fairly strict reliability, atomicity, and ordering guarantees: the servers cannot be guaranteed to be in the same state if they receive messages in varying orders, or if some of them receive messages that others do not.

Another variation is the client-server group. Clients multicast to the server groups, but the replying server multicasts the response back to the group consisting of all the servers plus the client. This is the pattern used in ISIS.

3

Using the Opentalk Communication Layer

The Opentalk Communication Layer provides a set of frameworks and components for use by protocol developers who are creating protocol layers in VisualWorks, operating on top of either the TCP/IP or UDP transport layers.

The Opentalk Communication Layer is designed to be lightweight but robust, easy to understand and easy to extend, and rich enough to support the tasks of protocol developers interested in creating protocols of any type. The Opentalk Communication Layer is, in particular, intended to support the implementation of protocols like RMI or IIOP on top of TCP/IP, stringent multicast protocols on top of UDP, and custom peer-to-peer protocols. It is also suitable for bulk data transfer protocols used to replicate memory and disk, and the continuous transfer protocols used in telephony, conferencing, and cooperative work applications, for digital audio and video. Opentalk follows the general outlines of the OMG architecture to enhance accessibility.

Apart from being a distributed component “construction kit,” the Opentalk Communication Layer provides a number of immediately useful components. There is a complete request broker implementation that supports configurations using several kinds of object adaptors that exploit either TCP or UDP sockets. Brokers can be configured to use standard unicast communication, or multicast and broadcast messaging. A set of basic services is also provided.

This chapter describes how to use these components in distributed applications. It provides several code examples, offering a pragmatic, rather than a theoretical, introduction to Opentalk.

Using a Broker

Any application wishing to send or receive remote requests needs to create and maintain a request broker. Request brokers provide transparent remote communication between Smalltalk images and represent the communication layer to communicating applications.

Creating and Configuring a Broker

Brokers exploit lower level network protocols to transport messages between Smalltalk images. For a broker to be able to receive remote messages it has to be associated with a specific *location* in the network, referred to as *access points*. Access points link the brokers with the underlying network protocol, therefore they are expressed in terms recognized by the protocol. This means that access points of TCP/IP based brokers are instances of IPSocketAddress.

Since a broker cannot function without an access point, brokers are always created at an access point. The accessPoint message can be sent to an existing broker to obtain the access point it is associated with.

Brokers are usually created using one of the instance creation messages implemented on the class side of BasicRequestBroker. In practice, you usually send these messages to class RequestBroker, which provides services that BasicRequestBroker does not.

For example, a standard Smalltalk-to-Smalltalk TCP broker is created on a specified port using the expression

```
RequestBroker newStstTcpAtPort: 4242.
```

This binds the broker to all available interfaces. The binding address is derived from the configuration of the host system that the Smalltalk image is running on.

To specify an address for the broker to a single interface, use the newStstTcpAT: variant:

```
RequestBroker newStstTcpAt:
   (IPSocketAddress hostAddress: #[211 25 29 171] port: 4242).
```

The creation method using full IPSocketAddress specification is useful when a broker has to run on a specific network interface on a host with multiple network interfaces. If the address is one of the host's real addresses, it will bind only to that address.

Also, for operating across firewalls, it is typically necessary to assign the firewall IP to the broker, and configure the firewall to forward messages on the port. The firewall and broker must both bind to the same port number. If the address is a firewall address, rather than one of the host's real addresses, object references will be exported with this address.

Both types of broker creation message are shortcuts for a configuration-based expression like the following:

```
(StandardBrokerConfiguration new
   adaptor: (ConnectionAdaptorConfiguration new
      transport: (TCPTransportConfiguration new
         marshaler: (STSTMarshalerConfiguration new))))
   newAt: anIPSocketAddress
```

Using configuration-based expressions allows one to tune runtime component parameters like timeouts and buffer sizes.

Starting and Stopping a Broker

An instance of a broker has to be activated to be able to mediate a remote communication. To bring a broker into an active "running" state it has to be started using message start. A broker can be stopped with the message stop; it closes all the open communication channels and deactivates the broker. A stopped broker can be restarted again with start.

```
| server client |
server := RequestBroker newStstTcpAtPort: 4242.
client := RequestBroker newStstTcpAtPort: 4243.
[   server start.
    client start.
    (client ping: (IPSocketAddress hostName: 'localhost' port: 4242))
       ifTrue: [Transcript show: 'Contact!']
       ifFalse: [Transcript show: 'Failure!'].
] ensure: [server stop. client stop].
```

Using Broker Events

Opentalk contains a generic event tracing mechanism. It is implemented by code that triggers normal events at significant points in the course of remote message sending, receipt, and execution. Some events are triggered in the request broker and others in the adaptor. (Adaptors can be used without a broker.)

A complete list of triggered events is captured in the class-side operationalEvents and errorEvents methods of both broker and adaptor classes.

Applications can register event handlers with brokers or adaptors using the standard event API, for example:

```
aBroker
    when: #importingReference:in
    send: #importingReference:in:
    to: anEventLogger
```

The parcel Opentalk-Core-Support provides useful classes that can be used to trace broker and adaptor events: ArgumentTransformer, EventCollector, and EventPrinter. Application may, of course, provide their own event handlers.

The following is an example using an ArgumentTransformer:

```
aBroker objectAdaptor
    when: #exportingObject:oid:in:
    send: #show:
    to: Transcript
    with: (ArgumentTransformer withBlock: [:args |
        'Exporting <1p> with oid <2p>.'
        expandMacrosWithArguments: args])
```

To register the same handler for a number of events the following pattern may be used:

```
log := EventCollector new.
#(#sendingRequest:in #receivingReply:in:) do: [:ev |
    aBroker objectAdaptor when: ev send: ev to: log]
```

There are additional convenience methods that support registering a handler for an entire class of events, like sendOperationalEventsTo:, sendErrorEventsTo:, and sendAllEventsTo:. For example:

```
aBroker sendAllEventsTo: EventPrinter new
```

Message Interceptors

Message interceptors (class MessageInterceptor) are a fairly common patternemployed by many middleware frameworks for distributed computing (e.g., CORBA) to allow applications to observe, and possibly intervene in, processing of remote messages.

While there are not many fundamental differences between message interceptors and broker event handlers, there are advantages to modeling these handlers as objects, rather than as ad hoc blocks

hooked into the broker events. For example, it is much easier to pass state from one interception point to another in the context of an interceptor object.

BasicObjectAdaptor maintains a ProcessingPolicy, which is configured as a processingPolicy aspect of AdaptorConfiguration. The main responsibility of ProcessingPolicy is to provide InterceptorDispatcher to any incoming/outgoing request.

By default, a new instance of InterceptorDispatcher, configured with fresh set of MessageInterceptor instances, is created to handle each request/reply pair. However, it is possible to optimize by reusing dispatcher/interceptor instances, if the processing is stateless.

The kinds of dispatchers and interceptors used depend on the policy that is currently configured with dispatcher class and a sequence of interceptor classes to use. These will be different for different protocols as the interception points might be different for different protocols. Users might or might not use their specialized class of dispatcher (if some pre/post processing is necessary before/after interceptors get invoked) and usually will provide custom interceptor classes filling in actual processing at specific interception points. Interceptors have a back pointer to their dispatcher which allows interceptors to reflect on other interceptors in case such coordination is necessary. Therefore the dispatcher also serves a a sort of "processing context" for the interceptors.

MarshalerConfiguration defaultProcessingPolicyClass defaults to the right type of interceptor or dispatcher to deal with protocol specific events. The policy default is delegated to marshaler configuration because it is the one configuration component that reflects the application protocol.

The same interceptor is expected to process the corresponding request and reply. Interceptors are assigned to a request as soon as it is created on the client or server side, before any interception points are reached.

Similarly, each reply gets the dispatcher from its corresponding request as soon as possible. On the client side the reply has to be first matched with the corresponding request. If the request is not found then a new dispatcher is obtained from the processing policy so that the interceptors can process events of the orphaned reply.

Remote Objects

This section discusses some concepts and inner aspects of distributed object computing that don't always surface in application development. However, you are guaranteed to see them sooner or later and it is very useful to have an understanding of these concepts ahead of time.

Now that we know how to setup a broker we can discuss how to actually use it for its primary purpose, remote object communication. The communication means remain the same, using normal Smalltalk messages. The complication is in how to address a message to something in a remote system. To send a message to a remote object, the object has to be identified using a kind of token that represents the object in the remote system. Surprisingly this token is called an object identifier or OID. Assigning an OID to an object is called *exporting* and an object is exported by sending message export:oid: to broker's object adaptor. An OID is usually a SmallInteger, but can be any kind of object that guaranties uniqueness of object of given value, e.g. Symbols can be OIDs but Strings cannot.

Once an object is exported a remote system is able to identify it using its OID. To send a message to an exported object on a remote system a broker method sendMessage:to: can be used. Its first argument is an instance of the Smalltalk class Message and its second is an instance of ObjRef. ObjRef is just a composite of OID and the remote system address, which in case of STST TCP request broker is IPSocketAddress.

```
| server client str size remoteStr |
server := RequestBroker newStstTcpAtPort: 4242.
client := RequestBroker newStstTcpAtPort: 4243.
[   server start.
    client start.
    str := 'Hello'.
    server objectAdaptor export: str oid: #Greeting.
    remoteStr := ObjRef newOnHostName: 'localhost' port: 4242
        oid: #Greeting.
    size := client sendMessage: (Message selector: #size) to: remoteStr.
    (size = str size)
        ifTrue: [Transcript show: 'Correct! ']
        ifFalse: [Transcript show: 'Incorrect! '].
] ensure: [server stop. client stop]
```

Of course it would be unpleasant if every remote call had to be expressed using sendMessage:to:. Therefore Opentalk provides a transparent wrapper for ObjRefs that does this for you. The wrapper is an instance of class RemoteObject. It's main purpose is to catch and redirect any message sent to it (using the usual doesNotUnderstand: trick), transforming it into a sendMessage: for the ObjRef it wraps. A RemoteObject may be obtained by sending the message remoteObjectToHost:port:oid:.

```
| server client str remoteStr size |
server := RequestBroker newStstTcpAtPort: 4242.
client := RequestBroker newStstTcpAtPort: 4243.
[   server start.
    client start.
    str := 'Hello'.
    server objectAdaptor export: str oid: #Greeting.
    remoteStr := client remoteObjectToHost: 'localhost' port: 4242
        oid: #Greeting.
    size := remoteStr size.
    (size = str size)
        ifTrue: [Transcript show: 'Correct! ']
        ifFalse: [Transcript show: 'Incorrect! '].
] ensure: [server stop. client stop]
```

This is how remote messages works internally. In everyday application development, RemoteObjects are usually obtained using higher-level services, and since they provide transparent messaging there isn't much OID and broker address juggling involved. We will talk about this more in the chapter on broker services.

Object Passing Modes

So far we've been sending remotely only very simple unary messages. However, messages can take a number of fairly complex parameters and return a complex object as a result as well. How do these objects get across to the remote system? By default they don't. ObjRefs are sent instead. This is called passing objects *by reference*. It means that if a parameter of a message is a complex object, and it wasn't exported yet, then it will be exported automatically in the course of message sending, and an ObjRef with its OID is sent instead of the object. The following code prints "Passed by reference!" to the Transcript:

```
| server client holder remote obj |
server := RequestBroker newStstTcpAtPort: 4242.
client := RequestBroker newStstTcpAtPort: 4243.
```

```
[   server start.
    client start.
    holder := ValueHolder new.
    obj := Object new.
    server objectAdaptor export: holder oid: #holder.
    remote := client remoteObjectToHost: 'localhost' port: 4242
        oid: #holder.
    remote value: obj.     "Pass the object across"
    (holder value _isRemote)
        ifTrue: [Transcript show: 'Passed by reference! ']
        ifFalse: [Transcript show: 'Passed by value! '].
] ensure: [server stop. client stop]
```

There are exceptions to this default behavior though. In general all immediate objects (nil, true, false, Characters and SmallIntegers), Magnitudes, ByteStrings, ByteSymbols, some collections, and others are sent across as is. (Actually their value is encoded, sent across, decoded and the objects are reconstructed from it on the remote system). This is called passing objects *by value* and the result of pass by value is usually 2 copies of the object: one original version in the local image and a new copy of it on the remote system. Keep this in mind because modifying an object passed by value does not affect the original at all!

To force an object to be passed by value, regardless of the default rules, use the result of sending asPassedByValue to the object. Similarly to force an object to be passed by reference, use the result of asPassedByRef. So if we slightly modify our previous example to use asPassedByValue, it will print “Passed by value!” to the Transcript instead.

```
| server client holder remote obj |
server := RequestBroker newStstTcpAtPort: 4242.
client := RequestBroker newStstTcpAtPort: 4243.
[   server start.
    client start.
    holder := ValueHolder new.
    obj := Object new.
    server objectAdaptor export: holder oid: #holder.
    remote := client remoteObjectToHost: 'localhost' port: 4242
        oid: #holder.
    remote value: obj asPassedByValue. "Pass the object across"
    (holder value _isRemote)
        ifTrue: [Transcript show: 'Passed by reference! ']
        ifFalse: [Transcript show: 'Passed by value! '].
] ensure: [server stop. client stop
```

For further discussion of pass modes, refer to Pass Modes.

Remote API of a Broker

Brokers themselves can be accessed remotely by any other object. A Broker automatically exports itself using a well-known name so that ObjRefs to it can be constructed on remote sites. This is done by sending an activeBrokerAtHost:port: message to the local broker, providing the remote broker's host name and port number as arguments. Alternatively activeBrokerAt: message with an instance of access point can be used.

```
| server client remote |
server := RequestBroker newStstTcpAtPort: 4242.
client := RequestBroker newStstTcpAtPort: 4243.
[   server start.
    client start.
    remote := client activeBrokerAt: server accessPoint.
    ((remote echo: 42) = 42)
        ifTrue: [Transcript show: 'Echo successful!']
        ifFalse: [Transcript show: 'Echo failed! '].
] ensure: [server stop. client stop]
```

Message echo: is part of the remote API allowing a test of the remote broker's responsiveness.

RequestBroker extends BasicRequestBroker with access control for the remote broker API. It maintains a list of so-called open selectors and allows only those selectors to be sent to the broker remotely. Open selectors list is maintained using messages openSelectors: and openSelectors. The list of open selectors is made use of in remotePerform:withArguments:, and it serves only to guard a broker against inadvertent tampering or damage. It does not safeguard other objects in the image, which are fully exposed to a remote user with a reference to them.

Broker Services

Another distinguishing feature of RequestBroker is support for broker services. RequestBroker maintains a simple registry of services keyed by service identifiers. A remote party can obtain a list of these service identifiers using the message serviceIds. Once the remote party knows the identifier of a service it would like to use, it can get a reference to it by sending message serviceById: to the remote broker passing the service identifier as a parameter. Message registerService:id: is used to add a service object to the broker's

registry. Any object can be a service. It can be something as generally useful as a naming service, or something tailored to the specific needs of a given application.

The reason why the broker services feature is so important is that it helps to resolve a well-known distributed object computing problem called initial reference acquisition. Once there are some remote references available, their set will grow quickly and autonomously as remote messages fly back and forth. The question is how to get the first remote reference in a freshly started distributed system.

The answer is to register the general entry point object under a well-known identifier as a broker's service. Then any remote system can access it using this identifier. Even if there isn't an established, well-known identifier, a remote system can discover the identifiers by asking for the list of services or service identifiers registered with a broker.

```
| server client remote |
server := RequestBroker newStstTcpAtPort: 4242.
client := RequestBroker newStstTcpAtPort: 4243.
[   server start.
    client start.
    server registerService: Random new id: #RandomGenerator.
    remote := (client activeBrokerAtHost: 'localhost' port: 4242)
            serviceById: #RandomGenerator.
    Transcript show: remote next printString.
] ensure: [server stop. client stop]
```

Opentalk Service

In order to facilitate service registration, Opentalk provides the abstract class OpentalkService as a convenience. It provides support for a default instance, a service identifier, and several methods for registering instances either

- in a naming service,
- as a request broker service, or
- in a broker's object adaptor under a OID specified by the user.

Concrete subclasses may invoke these methods in class initialization methods to ensure that default instances are registered in the Opentalk naming service or with a default broker at class initialization time.

The existence of the class OpentalkService is not a constraint; instances of any class may be registered as broker services, registered in a naming service, or registered in a broker under a specified OID. Note also that only those Opentalk services likely to have registered instances are subclasses of OpentalkService.

Using NamingService

A naming service is probably the most important distributed object service. Its basic function is similar to broker services registry discussed before, i.e. mapping well-known names to objects. Both are kind of a distributed equivalent of what name spaces are for a single image. The difference between the two is in their flexibility and robustness. While a broker service registry is very simple and flat, a naming service supports a hierarchy of naming contexts that can span multiple distributed images. A broker service registry is like a single name space called Smalltalk, whereas a naming service is comparable to the current hierarchical name space structure.

A hierarchy of naming contexts always needs a root from which the name resolution starts. A naming context is an instance of class NamingService and, as such, it is just another distributed object that needs to be exported so that it can be accessed remotely. Although it does not really matter how the root naming context is exported, common practice is to register it as a broker service. And indeed the class NamingService is a subclass of OpentalkService so that an instance of it can be conveniently created and registered using the service registration methods.

There is also a convenience method namingService that returns the registered context. If there isn't one registered yet, it creates one, registers and returns it.

The essential API of NamingService is comprised by the methods bind:to:, resolve:, resolve:ifAbsent: and unbind:. It was mentioned already that there often is a hierarchical structure of contexts. To resolve a name of an object registered deeper in the hierarchy a compound name can be used. A compound name is a sequence of names of contexts all the way from the context that is going to be asked to resolve the name (usually the root context) and ending with the name

of the target object. A compound name may be either a period-separated String or an OrderedCollection of names. Usage is demonstrated in the following example.

```
| server client remote |
server := RequestBroker newStstTcpAtPort: 4242.
client := RequestBroker newStstTcpAtPort: 4243.
[   server start.
    client start.
    "Enable remote use of #namingService method"
    server openSelectors: server openSelectors, #(namingService).
    "Create the root naming context and in it a nested context called
    Generators"
    server namingService create: 'Generators'.
    "Register an instance of Random as Random in Generators"
    server namingService bind: 'Generators.Random' to: Random new.
    "Obtain a remote reference to Random resolving its name in the remote
    naming service"
    remote := (client activeBrokerAtHost: 'localhost' port: 4242)
        namingService
            resolve: 'Generators.Random'.
    Transcript show: remote next printString.
] ensure: [server stop. client stop]
```

Using UcastEventService

Many applications exploit some kind of event-based mechanism to propagate notifications to interested parties. VisualWorks implements its own object level events (Object protocols “event *”), which are used throughout the system. Opentalk’s event services extend object level events with additional multiplexing components that are capable of relaying events to remote systems.

These event relays are instances of UcastEventService. A network of them constitutes a unidirectional distributed event channel. UcastEventService supports the standard VisualWorks event API implemented in Object protocols event *. In addition, there are a couple of configuration methods allowing you to plug UcastEventServices together to form a channel. These methods are addRelay:, removeRelay: and clearRelays. The following example will print “Hello!” twice to the Transcript in response to a single event triggered.

```
| b1 b2 b3 remote front back1 back2 |
b1 := RequestBroker newStstTcpAtPort: 4242.
b2 := RequestBroker newStstTcpAtPort: 4243.
```

```
b3 := RequestBroker newStstTcpAtPort: 4244.
[   b1 start.
    b2 start.
    b3 start.
    "Register the front relay of the event channel"
    front := UcastEventService new.
    b1 registerService: front id: 'channel1'.
    "Register back1 of the relay channel and plug Transcript into it"
    back1 := UcastEventService new.
    b2 registerService: back1 name: 'channel1'.
    remote := ((b2 activeBrokerAtHost: 'localhost' port: 4242)
            serviceById: 'channel1')
                addRelay: back1.
    back1 when: #show: send: #show: to: Transcript.
    "Register back2 of the relay channel and plug Transcript into it"
    back2 := UcastEventService new.
    b3 registerService: back2 id: 'channel1'.
    remote := ((b3 activeBrokerAtHost: 'localhost' port: 4242)
            serviceById: 'channel1')
                addRelay: back2.
    back2 when: #show: send: #show: to: Transcript.
    "And now try to trigger a #show event at the front"
    front triggerEvent: #show: with: 'Hello! '.
] ensure: [b1 stop. b2 stop. b3 stop]
```

Using a Broadcasting RequestBroker

The broadcasting RequestBroker exploits IP broadcasting as the underlying communication mechanism. Broadcasting, like multicasting, uses UDP sockets to deliver data packets. In broadcast, the recipients are determined by the form of the broadcast address. There are three commonly used broadcast address types:

Limited Broadcast

The limited broadcast address is 255.255.255.255. A packet sent to this address is never forwarded by a router. Thus, it never leaves the local subnet, but it is received by all stations on the subnet.

Net-Directed Broadcast

The net-directed broadcast address has a host ID (6 bits) of all one bits. A class A net-directed broadcast is of the form N.255.255.255, where N is the class A network number. All stations on the specified network receive the broadcast, unless

routers have explicitly disabled the default forwarding policy for such addresses. This is a wide area broadcast. It is expensive and seldom useful.

Subnet-Directed Broadcast

A subnet-directed broadcast address has a host ID (6 bits) of all one bits but a defined subnet ID. Routers determine whether a packet is a subnet-directed broadcast by reference to the subnet mask. This is the most common and most useful form of broadcast. It targets a specific network that may be different from that of the sender. However, routers may not forward these broadcasts either, in the interest of preventing denial of service attacks. This form of broadcast is only useful in a controlled environment, configured for broadcasts of this type. Any subnet-directed broadcast to the orioginating networl will have teh same effect as a limited broadcast.

A broadcast RequestBroker, by default, assumes that limited broadcast is desired. (See BcastTransport groupAddress, where this default is arranged.)

A broker may be configured to use net- or subnet-directed broadcast by sending BcastTransportConfiguration message networkDirectedWithNetmask:. The argument is a 4 byte ByteArray representing the netmask of a given network. It is used to compute the broadcasting address from the local host address. An equivalent message is networkDirectedWithNetmaskWidth:, which takes a number of non-zero bits in the netmask instead. Also, if a standard class C subnet is the intended target of the broadcast, the netmask can be set using networkDirected:. Finally, if a specific network configuration does not use the highest network address for broadcasting, a different address can be explicitly set with AdaptorConfiguration message accessPoint:. A typical creation expression for a network directed broadcasting broker looks something like this:

```
(BrokerConfiguration basic
   adaptor: (AdaptorConfiguration objectGroups
      transport: (TransportConfiguration bcast
         networkDirectedWithNetmask: #[255.255.254.0];
         marshaler: (MarshalerConfiguration stst))))
   newAtPort: 4242
```

Broadcasting is also targeted to a given port and only hosts reading given port will actually receive the packet. Therefore broadcasting RequestBrokers that are supposed to communicate together have to be running on the same port number.

Broadcasting is represented by the concept of object groups at the object level. In the case of a broadcasting RequestBroker it means that a given OID is not associated with a single object on a single host, but rather associates with a number of objects on different hosts. Therefore to build an object group with a set of broadcasting RequestBrokers each should export its group participant object using a given OID. This way an OID effectively becomes a group ID. To broadcast a message to participants in a group the message has to be addressed to a group proxy. A group proxy is obtained from a broker using groupById: message with parameter being that group's ID.

There's one more aspect of broadcasting that has a significant impact on usage of a broadcasting RequestBroker. Given the nature of broadcasting, where there is a potentially large number of recipients, it would be hard to support the usual two-way request/reply semantics of remote messages. How could one meaningfully combine a number of replies into a single return value? Therefore broadcasting RequestBrokers support only one-way requests, which means there's no meaningful return value from a message mediated by a broadcasting RequestBroker.

```
| b1 b2 b3 transcripts |
b1 := RequestBroker newStstBcastAtPort: 4242.
b2 := RequestBroker newStstBcastAtPort: 4242.
b3 := RequestBroker newStstBcastAtPort: 4242.
[   b1 start.
    b2 start.
    b3 start.
    "Export group participants with the group ID"
    b1 objectAdaptor export: Transcript oid: #Transcripts.
    b2 objectAdaptor export: Transcript oid: #Transcripts.
    b3 objectAdaptor export: Transcript oid: #Transcripts.
    "Create the group proxy using one of the brokers"
    transcripts := b2 groupById: #Transcripts.
    transcripts show: 'Hello ! '.
    "We need to give the brokers some time to process
    the incoming message before they are stopped"
    (Delay forMilliseconds: 100) wait.
] ensure: [ b1 stop. b2 stop. b3 stop ]
```

Using a Multicasting RequestBroker

A multicasting RequestBroker exploits IP multicasting as its underlying communication mechanism. Multicasting is very similar to broadcasting. Again it is a way to route a packet to a number of parties but this time the set of recipients can be fine-tuned, on a host-by-host basis. A packet to be multicasted has to be addressed to one of a range of dedicated multicasting addresses (224.0.0.0 – 239.255.255.255). The port number is important and has the same effect as described above for broadcasting. All that was said about broadcasting at the object level applies to multicasting as well: multicasts are sent to an object groups and are unidirectional.

```
| b1 b2 b3 transcripts |
b1 := RequestBroker newStstMcastAtPort: 4242.
b2 := RequestBroker newStstMcastAtPort: 4242.
b3 := RequestBroker newStstMcastAtPort: 4242.
[   b1 start.
    b2 start.
    b3 start.
    "Export group participants with the group ID"
    b1 objectAdaptor export: Transcript oid: #Transcripts.
    b2 objectAdaptor export: Transcript oid: #Transcripts.
    b3 objectAdaptor export: Transcript oid: #Transcripts.
    "Create the group proxy using one of the brokers"
    transcripts := b2 groupById: #Transcripts.
    transcripts show: 'Hello ! '.
    "We need to give the brokers some time to process
    the incomming message before they are stopped"
    (Delay forMilliseconds: 100) wait.
] ensure: [ b1 stop. b2 stop. b3 stop ]
```

Using McastEventService

McastEventService is one of the very natural applications of multicasting. Functionally it is almost equivalent to UcastEventService, i.e. it is an event relaying extension for VW object events. The difference springs from the nature of multicasting. Since multicasting is inherently omnidirectional, a network of McastEventServices forms an omnidirectional channel as well, unlike the UcastEventService which creates unidirectional channels.

McastEventService has to be used with either a multicasting or a broadcasting RequestBroker. To create a channel, simply form an object group with instances of McastEventService as participants.

Instead of maintaining a collection of event relays McastEventService just holds onto a group proxy, which is called its sender. To setup an instance of McastEventService with a group proxy use message sender:.

```
| b1 b2 b3 es1 es2 es3 |
b1 := RequestBroker newStstMcastAtPort: 4242.
b2 := RequestBroker newStstMcastAtPort: 4242.
b3 := RequestBroker newStstMcastAtPort: 4242.
[   b1 start.
    b2 start.
    b3 start.
    "Setup broker 1"
    es1 := McastEventService new.
    b1 objectAdaptor export: es1 oid: 42.
    es1 sender: (b1 groupById: 42).
    es1 when: #show: send: #show: to: Transcript.
    "Setup broker 2"
    es2 := McastEventService new.
    b2 objectAdaptor export: es2 oid: 42.
    es2 sender: (b2 groupById: 42).
    es2 when: #show: send: #show: to: Transcript.
    "Setup broker 3"
    es3 := McastEventService new.
    b3 objectAdaptor export: es3 oid: 42.
    es3 sender: (b3 groupById: 42).
    es3 when: #show: send: #show: to: Transcript.
    "And now trigger a #show: event at any broker"
    es2 triggerEvent: #show: with: 'Hello ! '.
    "We need to give the brokers some time to process
    the incomming message before they are stopped"
    (Delay forMilliseconds: 100) wait.
] ensure: [ b1 stop. b2 stop. b3 stop ]
```

4

Some Components of Opentalk

All three patterns of inter-process communication outlined in the previous chapter face common issues as a consequence of the fact that they all expect to send request identifiers, object identifiers, function identifiers, function arguments, and perhaps function definitions, within messages from one process space or host to another. As a consequence, the upper layers of all protocol stacks used for these messaging patterns tend to have common architectural components.

This section discusses some of those common issues and components, and points out where and how the logical components are represented in the Opentalk Communication Layer. Not all logical components are discussed, nor are all of the implementations of the Opentalk Communication Layer. Furthermore, no systematic attention has been devoted to the components characteristic of heterogeneous RPCs. More components, and components specific to heterogeneous RPCs, shall be addressed in future editions of this document.

Though it should not come as a surprise, additional information of the kind presented in this chapter occasionally may be found in the class comments of the Opentalk implementation.

Message Format

You cannot have a message without a message format. In practice a message format usually involves two separate chunks of specification: one for the message header and another for the message bodies of potentially several message types.

Message Header

A message header is seldom other than a fixed length byte array containing a fixed set of items each beginning at a predetermined position. The items are of a privileged sort, usually of the kind that determine whether and/or how the attached message body is to be processed. Reading the header is supposed to be cheap. It is standard practice in request processing to process the header, and then to branch on the outcomes.

For example, a recipient may defer processing the body based on a protocol version number. An older implementation of the protocol can usually assume that it is not forward-compatible with a later one. Revisions that do and do not retain forward-compatibility can be distinguished if a protocol version number has both a “major” and a “minor” component, where minor version number changes indicate forward compatibility and major ones do not.

An example of an item that affects the way in which the attached message body is processed is a message type identifier. For example, if the recipient gets a “time-profile” rather than a “normal” message, it may then know that it is required to run the message encoded in the message body under a time profiler, rather than in the ordinary way, and return the profiling results along with the usual return value, so that the sender can assemble a cross-platform execution tally.

TransportPackageBytes

In Opentalk, the header bytes of an encoded message are represented by the class TransportPackageBytes, and its potential subclasses. TransportPackageBytes is a subclass of BinaryStorageBytes with a default size of 12 bytes. It contains seven slots for representing:

- the “magic,” a protocol identifier
- the message type
- the major protocol version number
- the minor protocol version number
- the byte order
- an indicator of whether the message is a fragment
- the message length

Message Body

Receivers and senders have to agree on the ordering and the constituents of the potentially several message types. Message bodies may contain request identifiers, target object identifiers, function identifiers, arguments, and possibly other items relevant to the communication and execution options supported by the protocol. These will all be placed in the message body.

RemoteMessage and its Subclasses

Message bodies are represented in Opentalk by class RemoteMessage and its subclasses. The components of message bodies are held in instance variables. RemoteMessage itself is an abstract class and only declares one instance variable, for a request identifier—almost always useful.

Logical Message State Machines

In some protocol implementations, sending one logical message may involve sending several actual ones.

For example, one simple way to ensure consistent message ordering within a multicast group is to require all message senders to obtain a timestamp or sequence number from a common sequencing service, and include that timestamp in the header of any message they issue. (It is better, but still involves a single point-of-failure, if the message is unicast to the sequencer directly, where the sequencer attaches a timestamp or sequence number, and then multicasts the message to the group). In this fanciful protocol, sending one logical message involves sending two actual ones: one to the sequencer, then a second to the multicast group. With a sequence number attached to each message, each recipient can easily discover whether it is missing messages in its incoming queue, and delay further processing until the missing messages arrive.

In cases like this there will be some code component:

- that expresses the message ordering or the message exchange sequence required for completion of a single, logical message, and
- that records the location of the current logical message in the state transition diagram prescribed by the protocol.

See the discussion of session layers later in this chapter for more trenchant remarks on the issues associated with protocols involving complex exchanges.

Methods sendRequest:to and evaluateFor:

Opentalk has no dedicated class responsible solely for representing the relationship between a complex message and its component parts. Simple cases may be handled by overriding methods in the context stack proceeding from BasicRequestBroker>>sendMessage:to: on the client side, or in the stack proceeding from RemoteMessage>>evaluateFor: on the server side.

Note that these are also the two stacks wherein you are likely to place halts, whenever you need to examine either client-side message sending or server-side message receipt.

Server-Side Message Dispatch

Once a message has been received, the job of processing it has to be dispatched or assigned to some process. The style of dispatch is an important design dimension. One might defer dispatch and let the receiving process unmarshal and evaluate the message. This alternative is the least useful and the most dangerous: while processing the current message it will be available to receive others. It is more usual to dispatch the message to a newly created process running at a lower priority than the receiving process, or to dispatch it to a process in a pool of processes, also running at lower priority. Those processes in turn may evaluate the message and reply, or forward the message to yet another image. Regardless, every protocol will have some component that regulates server-side message dispatch.

The Methods handlingIncomingMessage and dispatchFor:

Opentalk does not have a dedicated class that handles message dispatch. Instead dispatch is regulated by handlingIncomingMessage, which in turn usually calls dispatchFor:. Both methods have several implementations in the Opentalk Communication Layer.

Process Environments

A message is sent from one process to another. The originating process may have environment variables that affect execution, and that must be copied to the environment of the process that will respond to the message on the remote host. Such process environment variables are items that are constant with respect to the logical, distributed process being executed. They retain their value irrespective of

- the host
- the number of hosts the that logical process has so far transited during the course of execution
- the concrete process currently executing
- the function currently being executed, or
- that function's arguments.

They may also be independent of:

- the current message type.

Such environment variables obviously cannot be treated in the same way as function arguments: they have far wider scope.

Two common examples of meaningful environment variables are security profiles and interface homes.

Security profiles are used in fine-grained security implementations, where users are granted or refused permissions at the level of whether they may send specified classes of messages to specified classes of objects. When a user logs on, his security profile is attached to the process environment of any process he or she initiates, and checked at each function invocation. If the user initiates a process that issues a remote request, the users profile must be copied to the environment of the process on the remote machine that handles the request, so that security constraints can still be enforced.

Interface homes are useful In VisualWorks, where several method implementations print messages to the Transcript or raise Dialogs. In a distributed environment, such implementations may be invoked during the course of a logically distributed process originating on another machine. In that case, the Transcript printed to should be the one on the originating image, and Dialogs should be raised at the

same location. So, the originating or home environment must be recorded in a process environment variable, and used to redirect messages sent to Transcript or Dialog.

Environment variables are seldom of concern to application developers, but are often critical in implementing distributed services. So, protocol layers above the transport layer will typically include environment variables as one of the components of a message body, and implement code that copies environment variables into message bodies from a sending process, and from message bodies to the process that responds to an incoming message. There should also be an API that allows service and application developers to define and add new environment variables.

Process Priorities

Priorities of the background processes created by Opentalk brokers are below the critical process priority range (LowIOPriority and above). Priority is configurable, allowing broker performance tuning.

The adaptor configuration parameters (ConnectionAdaptorConfiguration), which are applicable to connection-oriented adaptors, are:

listenerPriority

The listener process priority.

listenerBacklog

The number of allowed pending connection requests.

The transport configuration parameters (TransportConfiguration) are:

workerPriority

The worker process priority.

serverPriority

The server process priority

The serviceContext Instance Variable

Opentalk only defines service contexts for requests, as they only make sense for requests, and only in class STSTRequest. The method #sendAndWaitForReply: has the code that loads the sender's current process environment into a request. #evaluateFor: loads the service contexts of the incoming message into the process dispatched to

generate the reply. Objects are placed into the process environment by the sender using the method Process>>environmentAt:put: or by helper methods, also implemented in class Process, that invoke #environmentAt:put:.

Endianness

Communicating computers may not be of the same endianness. In a little-endian architecture the least significant byte of an integer has the lowest address. In a big-endian architecture, the least significant byte of an integer has the highest address. Any communication protocol must accommodate the fact that low-level data elements may have fundamentally different representations in the two communicating process spaces.

There are three common strategies for handling this issue:

- Assume that the communicating hosts will always be of the same endianness. This is rarely a viable option.
- Convert data values to and from a common external representation. This entails that you will always have work to do, on both the sending and the receiving side
- Transmit data values as is, with an architecture marker, so that the receiver of the data can convert the data as needed. This is the lightweight option. In this option, the architecture or endianness marker is one component of the message header.

Thus, any generally useful protocol stack implementation will involve either a common data representation or an endianness-marker-and-switcher.

The byteOrder and swap Instance Variables

Opentalk opts for the third design option described above. It records endianness in the byteOrder instance variable of MessageHeader. MessageHeaders understand the message #swap, the return value of which is used to set the boolean value of the swap instance variable of STSTStream. The value of that instance variable is referred to by several unmarshaling methods: #nextDouble, #nextFloat, #nextLong, and their like. The swap instance variable only records whether STSTStream should swap byte order while unmarshaling. The local endianness is recorded in the isBigEndianPlatform instance variable of STSTStream, and is used in the message #nextLong.

Encodings

In heterogeneous environments, communication occurs between processes implemented in languages that have different native character set encodings. Note that this is a translation problem, specific to character data, at the level of the implementation language rather than the host.

The options for handling this problem are the same as those for handling the endianness issues—where assumptions about, or markers for, the encoding replace those for endianness—but with one customary addition. Senders and receivers may negotiate, usually just after the connection has been made in connection-oriented transfer protocols, whether the server or the client will do the work of translating encoded data between the client and the server encodings.

This option is one example of a non-trivial message sequencing issue: a negotiation protocol becomes one component of the overall communication protocol. In cases where servers have significantly more power or lighter loads than clients, “server does the work” will be favored over “client does the work.”

Whenever this issue arises, there will be some code component that specifically addresses encoded data. There also will be machinery for marking which items are to undergo encoding translation and which are not. This is the rational behind the distinction between the “character” and “octet” data types in the OMG IDL specification: characters undergo encoding translation, but octets do not.

Character value: self nextLong

Opentalk does not currently have machinery for handling character encodings, because its existing concrete protocols assume a homogeneous Smalltalk environment. But it does linearize characters as SmallIntegers, in anticipation of future encoding machinery.

Marshaling and Unmarshaling

Data is usually structured: Smalltalk objects are trees of lesser objects. But the data in a message packet is sequential. As a consequence of this mismatch, any generally useful protocol stack must provide facilities for flattening or linearizing data structures or object trees before transmission and rebuilding them from their

linearized form after receipt. So, even if you do not need a common data representation to handle the endianness or the encoding issues, you will wind up with one to handle the linearization problem: you cannot escape work on both the client and the server side.

Marshaling is the process of linearizing object structures into basic data elements, and converting the latter to a common data representation; unmarshaling is the reverse process of translating from the common data representation.

In the implementation of a protocol stack, the components for handling endianness, encoded data, and linearization will usually be integrated in a marshaling component.

STSTStream

Opentalk uses a stream class for marshaling and unmarshaling. The main entry point in its API are #marshalObject: and #unmarshalObject. The marshaling and unmarshaling machinery sensitively depends on the tables implemented on the class side of STSTStream. STSTStream must and does interact heavily with the pass mode control machinery discussed in Pass Modes.

STST marshaling employs a set of class-specific type tags, which allows optimized marshaling of several commonly-used classes. Several blocks of type tags are reserved for customer use, so that customers may implement and test application- and class-specific marshaling optimizations of their own. We assume no responsibility for conflicts in the use of the tag values within those reserved blocks (see the initializeTagDispatchTable class method in STSTStream for block identification).

Object References

Distributed systems require object references. In RPCs, object references represent a particular remote object. In group multicast, they represent a particular multicast group.

An object reference contains the information needed to establish a connection to the process space in which the remote object resides. In practice, an object reference contains at least two pieces of data:

- an access point for connecting to a remote application, and
- the unique identifier of the object that a reference points to.

The nature of both the access point and the object identifier depends upon the type of communication protocol being used. Some protocols will require additional data elements

An object identifier is set by the application from which the reference is obtained. The client application that obtains the reference is interested in the object identifier solely as a possible means for maintaining the local identity—rather than equality—of references pointing to the same remote object.

ObjRef

In Opentalk, object references are implemented by class ObjRef. This class may be subclassed as required. ObjRef is a subclass of class Object because it is a mere data holder. ObjRef does not reimplement doesNotUnderstand:, and instances of the class are used primarily as components of proxies and as indices in object tables. Object references are passed by value.

Like several other critical types, ObjRef overrides #marshalWith:. This double dispatching of the marshaling operation makes it easy to splice a custom marshaler into the Opentalk framework for objects that demand custom care.

Transparent Forwarding

The standard implementation of transparent forwarding in Smalltalk involves a proxy. A proxy is an object that stands in the place of another, usually remote object, which it represents. The class of the proxy object reimplements #doesNotUnderstand:, to forward incoming messages to the remote object represented. The proxy class usually inherits directly from either nil or class Object, so that there are few messages that it does understand.

Several kinds of proxy classes are possible and useful. The most usual are:

- the simple forwarding proxy just described, and
- the faulting proxy.

In response to the first message sent to it, a faulting proxy replaces itself with a copy of the object it represents and redispatches that first message to it. Subsequent messages arrive at the copy directly.

In Opentalk, the notion of a proxy is implemented using two classes, Proxy and RemoteObject. Faulting proxies are not now implemented. However, pass mode wrappers, to be discussed later, are treated as kinds of proxies.

Proxy

Proxy is an abstract class that inherits from nil and is intended to be the superclass of all of the several kinds of proxies. Proxies are wrappers of object references, and hold an instance of class ObjRef in their single instance variable. This design allows one to treat remote object access data and forwarding or faulting strategies orthogonally.

Class Proxy implements several messages whose selectors begin with an initial underscore. The underscore indicates that these messages will be evaluated locally and will not be forwarded.

RemoteObject

A concrete implementation of a forwarding proxy, RemoteObject, adds a relationship to a request broker to the Proxy behavior. This relationship is important for both sides. RemoteObject needs a broker to be able to invoke a remote request. Conversely, a request broker needs to keep track of imported remote references and corresponding RemoteObjects.

Object Reference Equality

It is possible for an image in remote communication with another to obtain a reference to the same object more than once, either as a return value in a reply or as an argument in a request. If the image retains those several references, it will significantly complicate its programming semantics if they are not identical. Furthermore, the test for identity should be as cheap as usual; it must not be a remote call. Thus, any good implementation of object references will ensure that references to the same object are identical, and that their identity is evaluated locally.

BasicRequestBroker's remoteObjectRegistry

In Opentalk, all request brokers inherit from BasicRequestBroker. That class has an instance variable called remoteObjectRegistry that holds a weak value dictionary: its associations are garbage collected when there are no longer any strong references to the value. The routines for unmarshaling object references all eventually call objectByRef:. If

the reference is not a reference to a local object that has round-tripped, a new local instance of class RemoteObject is found or created to represent it. The method findOrCreateRemoteObject: ensures the local identity of object references.

This implementation entails that references are identical only with respect to the broker through which they have been obtained. In practice, this is not a troublesome constraint. Note that if you intend to simultaneously support several types of proxies in the same broker, you will need to enhance the machinery that maintains reference identity.

Object Tables

Object adaptors require object tables to keep a record of the object references they export. The objects are indexed under the object identifier (OID) assigned to them by the broker or table. This is the identifier included in a reference, to uniquely identify the object among those exported by users of the table. When the adaptor processes an incoming request, it looks up objects under the OID to identify the object that is the target of the request.

Object tables must also support indexing of OIDs under objects. This lookup is employed whenever the adaptor creates and exports a new reference, to determine whether a reference to the object has already been exported and a pre-existing OID should be reused.

Though necessary to the functioning of an object adaptor, and for ensuring a stable association between the OIDs found in messages and the objects they refer to, object tables should be thought of as generic components. Resource managers of several kinds may need to maintain an association between OIDs and the objects within the resource manger's domain. Thus, you may have need for and use object tables in the implementation of several distributed system components.

There are three major design issues involved in implementing object tables.

- Object tables must be fast. Object and ID lookups are fundamental to message processing, and persons profiling a distributed system should have the confidence entailed by knowing that the cost of these lookups need never be their concern.

- A decision needs to be made about the stability of the object table. Ideally, it should be stable across image shutdown and startup, so that references exported before a shutdown or crash remain valid after the image has been restarted.
- A decision needs to be made about the strength of the references to objects within the table. If the table uses weak key and weak value dictionaries, table entries referring to an object will be garbage collected once the object is. If the usual, strongly referencing dictionaries are employed, the mention of an object in the object table will protect it from garbage collection.

It is standard practice to use weak key and value dictionaries in the implementation of a request broker so that the issues involved in retaining references to exported objects are addressed, as they usually should be, at the application rather than the communication layer. However, it may make sense to use strong references in an object table employed by a resource manager.

ObjectTable

Object tables are implemented in Opentalk by class ObjectTable. ObjectTables have four instance variables: an access lock, a weak key dictionary, a weak value dictionary, and the value of the numeric OID to use, on creation of a new entry. The central public protocol consists of three methods: #export:, #export:oid:, and #objectByOID:. Object adaptors retain an ObjectTable in their objectTable instance variable.

Opentalk's ObjectTable is now hard-coded to use weak key and weak value dictionaries. A future release will support the option of creating ObjectTables with strong references.

Request Brokers

A request broker is the primary representative of the distribution framework to an application. Its main purpose is to mediate access to all the distribution services, and their corresponding APIs, that an application may need. An application sets up an instance of the distribution machinery by creating an instance of request broker. It activates or deactivates the machinery by starting or stopping the broker. Opentalk's brokers also have built-in capability for automatic stopping and starting when the smalltalk image shuts down or starts up: a broker that was on when an image is shut down will restart when the image starts up.

Request brokers support two styles of remote request invocation. There is an RPC style API consisting of messages #sendMessage:to: and #sendMessage:to:timeout:, that allows explicit invocation of remote request. The second style allows application developers to use the same means, Smalltalk messages, for both local and remote communication. It is provided by a transparency layer of RemoteObjects wrapped around the RPC style API.

All the above is implemented in class BasicRequestBroker. It's subclass RequestBroker extends it with additional features like remote access control to broker APIs through the list of open selectors and explicit support for broker services. Broker services are a great help especially with the initial phases of distributed application start up. To be able to obtain contact with application components at startup, a component needs to obtain an initial set of references to relevant root objects that will support access to, or discovery of, other remote objects. A broker services registry plays an important role in this task. It allows one to register a service under a well known name using #registerService:id:. Once a service, for example, a naming service, is registered, other systems can retrieve remote reference to it using a known broker method (#serviceById:) and the well-known service name. A broker also allows retrieving names of all its registered services using the method #serviceIds.

Object Adaptors

Object adaptors represent the hidden side of the distributed framework. As opposed to brokers, which are mostly responsible for presenting the communication layer to an application or image, adaptors are there to talk to the outside. An adaptor manages a specific protocol implementation and maintains the necessary infrastructure needed to dispatch incoming and outgoing remote messages.

An adaptor uses low level network protocols to transport encoded requests to the target system. Network protocols use their own notion of location to be able to target the delivery of data. Each location has its own identifier. In the case of the TCP/IP protocol suite, this identifier is called an IP address. Opentalk uses the more generic term “access point” to leave room for other protocol suites. An object adaptor is instantiated for the access point that is provided by the broker that creates the adaptor. That's why the broker creation protocol requires an access point specification as a parameter.

Access points allow for dispatching requests to specific systems. However, a distributed object framework needs a finer addressing scheme to be able to dispatch requests to individual objects within the systems. Therefore object adapters export objects with their own identifiers (OIDs), which are then used as part of the addressing scheme represented by object references (ObjRefs). Adapters need to maintain a mapping between OIDs and objects that they export. This mapping is represented by an object table.

Adaptors exploit network protocols to conduct communication over the network. Generally, there are two fundamental classes of protocols with different usage patterns; therefore we have a different adaptor for each protocol class. First, there is ConnectionOrientedAdaptor for connection-oriented, “telephone-style” protocols like TCP. For these protocols two parties have to establish a connection before they can begin to exchange data. Second, there are connection-less “mail-style” protocols represented by ConnectionLessAdaptor. With these protocols the sending party attaches an "address" to the data and releases it to the network. The data later arrives to the recipient, unless it gets lost or the recipient is not watching for any data to come. A typical example of this type of protocol is UDP.

Connection-oriented protocols need some additional infrastructure for establishing and maintaining connections. The standard connection creation technique involves a “listener.” The side initiating the connection is usually called the client and the other side the server. Clients make special connection requests. The sole purpose of a listener is to wait for them, and pass up to the server, any connection creation requests that are delivered to server's access point. Therefore each active instance of ConnectionOrientedAdaptor maintains a running instance of ConnectionListener fulfilling this role. Once a connection is established it is registered in the adaptor's connection registry. When the connection is closed it is deregistered.

Transports

An object adaptor needs to maintain certain amount of networking infrastructure for the network protocol that it exploits. Apart from that there is also encoding and decoding machinery (marshalers) and process handling components. All this is expressed in a form of Transport in Opentalk. Transport implements an actual transport protocol. In order to track request execution status, each outgoing request is given its request ID and is registered in the Transport's

requestRegistry. The request ID is used to match incoming replies to the requests that they originated from. Once a reply arrives the corresponding request is unregistered from the registry and the reply's value is returned as a result to the process that initiated the remote request.

Incoming requests are intercepted and dispatched by the transport's server process. The server process is started upon transport creation and terminated upon transport shut down. The purpose of the server process is to recreate the request on the server side and dispatch it to the appropriate receiver. There is no client process because request encoding and sending is performed within the context of the process initiating the remote request invocation. Note that the server process is not responsible for request evaluation and reply send-out, because time consuming requests could significantly reduce the system's responsiveness, or even cause request losses in case of unreliable network protocols.

Connection-oriented protocols are implemented as StreamTransports. A StreamTransport represents one end of a lower level network connection. Instances of StreamTransport are registered by ConnectionOrientedAdaptor as connections.

Connection-less protocols are implemented as DatagramTransports. ConnectionLessAdaptor maintains a single instance of DatagramTransport for all of its interaction with the network.

Pass Modes

The arguments in a request and the return value of a reply may be passed in any of the three usual ways:

pass by value

In pass-by-value, a copy of the object is transmitted from one process to another. It presupposes that both the sender and receiver have equivalent implementations of the object in question.

pass by reference

In pass-by-reference, a reference to an object in the object space of the sender is transmitted to the receiver.

pass by name

In pass-by-name, only the name of a named object in the object space of the sender is sent to the receiver. It presupposes that the receiver has an equivalent object in its object space under the same name. The receiver uses that identically named local object in place of the name transmitted by the sender.

Though these are well-worn alternatives to the designers of programming languages, they have added dimensions in the realm distributed computing. To these three, Opentalk adds:

pass by OID

To improve efficiency, some distributed applications pre-replicate selected objects to all involved locales. In such cases, if a replicated object is an argument to a remotely invoked operation, it is a waste of resources to pass the replicate by either reference or value.

Any message sent to a reference immediately involves network traffic. Any message sent to a copy, which has been passed by value, has no network costs and, without additional machinery, any change is not propagated back to the original. Any name passed between images is usually assumed to refer to an identical implementation at both locales. This assumption is easily violated, and the consequences of a violation may be extremely difficult to debug.

Pass-by-OID allows pre-replicated objects to be passed by no more than the object identifier (OID) under which they were pre-registered in the object tables of both the sending and the receiving object adaptors. A passed-by-OID object, on receipt, resolves to either (a) its local replicate, or (b) an exception if the passed OID has not been pre-registered at both sending and receiving locales. You may think of pass-by-OID as a species of 'pass-by-name' for domain class instances.

Classes, NameSpaces, NameSpacesOfClasses, BindingReferences, LiteralBindingReferences, and Signals may be passed by name using, for example,

```
#{Object.DependentsFields} asPassedByName.
```

On receipt, a passed-by-name object will resolve to either (a) the local object that bears the passed name, or (b) an exception if there is no such object. The default pass mode for the six classes mentioned remains #reference. This is because use of pass-by-name

should be explicit; pass-by-name is reliable and straightforward to debug only when identity (or scrupulously calculated divergence) or implementation is ensured.

Pass modes have such immediate repercussions and costs—either in terms of network traffic, state replication, or the maintenance of implementation identity—that no facility for remote messaging is complete unless it provides pass mode control. Designers of distributed systems deeply care about, and need to be able to affect, the way that the arguments and return values of requests and replies are passed.

There are three major design decisions involved in implementing pass mode control:

- You must decide whether to support pass-by-name. The usual decision is to decline. It is just too fragile in a distributed environment.
- You must decide on a set of default pass modes, so that users are not required to make pass mode decisions with inconvenient regularity and so that the defaults protect users from unpleasant surprises.
- You must decide on what levels to support control. In an object-oriented language this devolves into a finite set of subsidiary decisions about whether to support control:
    - over all instances of a class,
    - by a class over the pass mode of its instance variables, and
    - at the level of a single object.

Your aim is a complete set of safe defaults, the definition of meaningful levels at which defaults can be overridden, and an obvious interface for both setting and overriding defaults. Almost by necessity, the implementation of a pass mode control system is spread about, and manifest in several different parts of the system.

Pass Mode Control

In earlier versions of Opentalk, when there were only two available pass modes—by value and by reference—the method isPassedByValue was overridden to set the default pass mode of an object. This is no longer true. Current Opentalk users should pay

special attention to this fact. Unmodified user implementations of isPassedByValue may not have their intended effect in the new version of the STST protocol. The methods

```
isPassedByValue
isPassedByReference
isPassedByName
isPassedByOID
```

are now used *only* to test the pass mode of an object or a PassModeWrapper. An object's default pass mode is now changed by overriding the method passMode.

A particular instance's pass default mode may be superseded by folding the instance in a PassModeWrapper. This is accomplished by sending the instance any one of the following messages:

```
asPassedByValue
asPassedByReference
asPassedByName
asPassedByOID
```

If an instance already sent by value is sent asPassedByValue, the instance rather than a PassModeWrapper is returned. The same holds for the other three methods. If an instance's default pass mode cannot legally be superseded by the desired pass mode, a pass mode exception is raised.

SpecialTypeDispatchTable and TagDispatchTable

STSTStream optimizes the pass mode of a few special objects—nil, true, and false—in its dispatch tables. They are, in effect, represented by integers in message bodies. Users cannot easily override the pass mode of these objects, and they should not. The local nil is just as good as any remote one, and costs a lot less to talk to. UndefinedObjects and Booleans are paradigmatic immutable types.

isPassedByValue

Other default pass modes are established by several implementations of #isPassedByValue, located at critical points in the class hierarchy. Note that BlockClosures are passed by reference: you cannot assume that a remote image contains a lexical closure's environment of definition.

asPassedBy methods

The methods #asPassedByRef, #asPassedByValue, #asPassedByName, and #asPassedByOID are all implemented in class Object, can be sent to any object to override its default pass mode.

PassModeWrapper

The methods #asPassedByRef and #asPassedByValue wrap an object in a PassModeWrapper as required. Pass mode wrappers are implemented as subclasses of Proxy because they have similar semantics: a PassModeWrapper should neither look nor behave like a local object; it is a local object on its way to being a remote copy or reference.

passInstVars and PassModeTable

The method #passInstVars may be implemented in any class, allowing that class to override the default pass mode of the objects contained in its instance variables. The method has only one implementation, in class Object, where it answers an empty array. Substantive implementations will answer an array, usually of the same size as the number of instance variables in the class. The array will contain symbols, one for each instance variables. The symbols are:

#true, to pass an object using its default pass mode,

#false, to pass a nil,

#ref, to pass by reference, and

#value, to pass by value.

The method passInstVars is called by STSTStream>>nextPutObjectInstVars:. The implementation makes use of STSTStream's PassModeTable for operational dispatch during marshaling.

inspectorClassName

Pass-by-name is employed in Opentalk only to pass the names of inspector classes. We can assume that these classes are uniformly implemented. In this instance, pass-by-name is enforced by several implementations of the method #inspectorClassName, and overrides of the method #inspectorClass, found in the parcel Opentalk-Core-Support.

Special Implementation of Behavior class>asPassedByValue

Early in the design of Opentalk, we experimented with using pass by name for classes. This turned out to be far too sensitive to implementation differences between communicating images to be of general use. Now, classes are passed by reference. Furthermore, Behavior class>>asPassedByValue is implemented so as to discourage passing classes by value.

Though we discourage pass by name, you may disregard us. For example, with a very small amount of work, you may implement a subclass of GeneralBindingReference that gives you pass by name for classes, name spaces, and shared variables.

Exceptions

Distributed applications run in a fragile environment. Hosts and network connections go up and down. The application needs to be notified of such events to be able to cope with them in a discriminating manner, that supports the requirements of the application. That's why distributed frameworks generate a fair number of exceptions that they do not handle.

Opentalk's exception set can be expected to dynamically evolve along with Opentalk itself. New types of exceptions will be added with new protocol implementations. On the other hand, more sophisticated protocols can intercept and handle some of the existing exceptions internally.

OtException

All exceptions raised by the Opentalk Communication Layer, and the other layers of the Opentalk system, are subclasses of OtException. OtException is an abstract class with three abstract subclasses: OTSystemException, OtServiceException, and OtComponentException. OtSystemException is the abstract superclass of all concrete exceptions specific to the Opentalk Communication Layer. The other two are the abstract superclasses of concrete exceptions respectively specific to Opentalk services (like NamingService) and Opentalk components (like service brokers).

OtSystemException and its Subclasses

All exceptions specific to the Opentalk Communication Layer are subclasses of the abstract exception, OtSystemException. OtSystemException has several concrete subclasses.

OtECommunicationFailure covers a fairly wide range of low level (socket) errors or expired timeouts. More detailed explanation of the encountered problem is provided as its messageText.

OtETimeout is a subclass of the generic OtECommunicationFailure that allows the timeout exception to be resumable. When resumed, it makes the client thread wait for another timeout period. This provides better support for long computations that may exceed the default timeout settings.

OtEInvalidObjectReference is signaled when a request receiver is not found in the local system when evaluating a request. This usually means that the OID of the target ObjRef is no longer valid. The most common reason is that the target object was garbage collected.

OtEMarshaling covers any kind of problem that occurs during request marshaling and unmarshaling.

OtEServerError is raised on the client side when an application error occurs during a remote request evaluation. The remote exception is forwarded to the client in an STSTErrorReply. The forwarded error cannot be resumed.

Catching Broker Errors

In the server process loop by default we handle any Error, generate an error event and resume the loop expecting next message. However by default nobody is watching broker events, so server process errors often are silently suppressed and the user sees only side-effects of those, such as timed-out requests or requests returning with “Connection Closed” in case the client or server broker is shuting down early enough.

To report such errors, an event handler has been added that uses the broker event mechanism. The API for handling broker errors is:

handleErrors (default)

Uninstalls any event handlers added by the other methods, restoring the default behavior or suppressing unexpected errors.

passErrors

Do not handle unexpected exceptions, but passes the exception.

haltErrors

Makes the error handling process loop halt on an unexpected error event.

showErrors

Prints unexpected error events in the Transcript

Refer to the method comments in BasicRequestBroker for more information.

Session Layers

One design decision affecting protocol implementations is whether to include a conversational or session layer above the connection layer and below the adaptor layer.

A session layer is commonly used to implement dialog control. Session layers may be used for token management, when the communicating parties are each forbidden to engage in the same operation at the same time. They may also be used for synchronizing and check-pointing long-running transfers, liable to interruption by link failures or crashes.

In practice, session layers have scant use in most applications, and it is doubtful that a session layer has any place in any general-purpose communication framework.

A standard critique of the OSI Reference Model is that it defines several layers that in practice are non-existent or thin. Its session layer is the one usually cited for its absence. The British proposal for OSI had five layers, not seven. The general consensus is that a truncated OSI model, sheared of its presentation and session layers, is useful for discussing computer networks. On the other hand, while the TCP/IP model is little better than an afterthought, its widely useful implementation has never had a session layer. Commonly used hybrid models ignore the session layer entirely.

Session layers are a possible design option in protocols that require multiple round-trips. If your protocol involves several complex negotiations—in other words, if it is highly stateful, because you need to remember where you are—then it is not irrational to consider a

session layer. It just happens to be far better, and far more flexible, to push the required conversational state into the objects communicating, or used to communicate, than to hardwire the notion of a session into the communication framework itself.

STSTRequest

Fundamentally, a session is just a stateful structure waiting for a return. In its Smalltalk-to-Smalltalk protocols, Opentalk implements this in the request itself, which contains all of the fundamental conversational data: target, request ID, message, reply, and so forth. Note that the logic of a message type, which may be arbitrarily complex, does not get marshaled.

This approach keeps the communication layer simple, speedy, and light. It also provides a strong hint about the right way to add more complex, special purpose protocols to Opentalk.

5

Broker Configuration

All of the supplied methods for creating a request brokers are implemented using configurations. Because the supplied set of broker creation methods is sufficient to create all the most commonly used brokers, most users can ignore the underlying Opentalk configuration framework. However, if you need to tailor a broker for some special purpose, knowledge of the options provided by the broker configuration system is essential.

A configuration is a blueprint for the creation of an object. It is an object in its own right, independent of the objects created from it. Thus, a configuration can be reused in two ways:

- A configuration can be stored, for example, in a method, and reused to create several objects with the same configuration.
- A configuration can be stored in the object created from it, making it easy to restore that object to its original configuration.

A separate blueprint is useful whenever the kind of object to be created is complex, structured, and amenable to being customized in several ways. Brokers are such things, and that is why configurations are used to create them.

Standard Broker Creation Methods

All of the standard broker creation methods on the class side of BasicRequestBroker are implemented in terms of configurations. For example, newStstTcpAt:, shown below, explicitly creates a nested configuration. It then sends the method newAt: to the created BrokerConfiguration to create a broker instance:

```
newStstTcpAt: anIPSocketAddress
    ^(StandardBrokerConfiguration new
        adaptor: (ConnectionAdaptorConfiguration new
            requestDispatcher: RequestDispatcherConfiguration standard;
            transport: (TCPTransportConfiguration new
                marshaler: STSTMarshalerConfiguration new
                )))
        newAt: anIPSocketAddress
```

The configuration configuration options are represented by classes in the Configuration hierarchy.

Notice that the broker configuration created in newStstTcpAt: has five parts:

- an enclosing BrokerConfiguration
- an AdaptorConfiguration
- a RequestDispatcherConfiguration
- a TransportConfiguration
- a MarshalerConfiguration

In its role as the blueprint for an object, a configuration's structure indirectly reveals the structure of the object created from it. The structure of the broker configuration method spells out the fundamental components of a broker, which consists of:

- A broker's API, standard or otherwise, that is the interface useful to the application making use of remote communication.
- An adaptor, mapping Smalltalk messaging semantics to a major protocol family.
- A request dispatcher that assigns the handling of incoming remote requests to local Smalltalk processes.
- A transport that implements the interface to a transport layer like TCP or UDP.

- A marshaler that translates Smalltalk objects into and from an on-the-wire encoding, like STST or CDR.

Because both brokers and their configurations exhibit the same layering, it is straightforward to specify a broker using configurations.

The Configuration Classes

All configuration classes inherit from class Configuration. The Configuration class hierarchy is defined so as to parallel the hierarchies of the configured broker and component classes. This is shown by the two hierarchies reproduced below. The first hierarchy is for request dispatcher components. The second is for the request dispatcher configurations used to create request dispatchers.

```
Object
   EventManager
      GenericProtocol
         RequestDispatcher
            HighLowRequestDispatcher
            PoolRequestDispatcher
            StandardRequestDispatcher
Object
   Configuration
      RequestDispatcherConfiguration
         HighLowRequestDispatcherConfiguration
         PoolRequestDispatcherConfiguration
         StandardRequestDispatcherConfiguration
```

In some component and configuration hierarchy pairs, the structural parallelism, or the match between the two sets of class names, may not always be as thorough as in this case. However, there is a configuration class for every concrete broker or broker component class.

Configuration Instance Variables

The instance variables of a configuration class holds either a configurable parameter or the configuration of a subcomponent. A good example of this is class BrokerConfiguration, which inherits or declares four instance variables:

- autoRestart, an optional Symbol determining when the broker should automatically restart
- id, an optional Symbol uniquely identifying the broker

- adaptor, a required AdaptorConfiguration
- requestTimeout, an optional integral number of milliseconds

In this case, three of the variables hold a configurable parameter and one holds the configuration of a broker sub-component.

After a configuration is created, methods are sent to it to set configurable parameters or configurations. You only need to set a configuration instance variable if you intend to use a non-default value. This is a product of the way in which components created from configurations are designed, particularly of the way that components lazily access the values in their configuration.

The Component Classes

The components created from configurations support an API that supports such creation. That API follows a standard pattern.

Component Instance Variables

All of the components created from configuration declare or inherit a configuration instance variable for storing their source configuration. On the class side they implement a new: method that takes a configuration as its argument. This method calls an instance side initialize: method, that takes the same argument. At its beginning, an initialize: method stores the configuration. Thereafter, it executes any required setup code.

Configuration Defaults

The default values of configurable parameters are stored on the class side of configurable objects. Their value is almost always controlled using a standard set of three methods

- a method with the prefix ‘default’ and the suffix ‘Value’ that establishes the default value of shipped code
- a method with the prefix ‘default’ followed by the parameter name that accesses the current default value
- a method with the prefix ‘default’, followed by the parameter name and a colon that changes the default value

Class BasicRequestBroker, for example, implements:

```
defaultRequestTimeoutValue
    ^60000
```

```
defaultRequestTimeout
    ^defaultRequestTimeout ifNil: [
        self defaultRequestTimeoutValue]

defaultRequestTimeout: aSmallInteger
    defaultRequestTimeout := aSmallInteger
```

The first sets the base default to 6000 milliseconds. The second accesses either the base default value or a value set by the user. The third is used to reset the default. A like set of methods is implemented for most configurable parameters.

Default Accessing

Components always access their defaults lazily, first checking their configuration, then checking their class-side defaults. Because of this, configurations may always override the defaults set by a component class, but components need not specify values that do not need to be changed from the default.

In those cases where the value may usefully be changed at runtime, at the instance level, the configured component also declares an instance variable for the parameter. In that case, the instance variable is checked before the value present in the configuration or the value set as the class-side default. The accessor requestTimeout in BasicRequestBroker illustrates this latter pattern:

```
requestTimeout
    ^requestTimeout ifNil: [
        configuration requestTimeout ifNil: [
            self class defaultRequestTimeout]]
```

Configuration Types

This section lists all the methods implemented in the “types” protocol of the abstract broker or broker configuration classes. Each method creates an instance of a concrete configuration class. When a component is produced from that configuration instance, it will be an instance of the componentClass that the configuration specifies.

In the following, each method is listed under the class that implements it. Each method name is followed by the name of the componentClass that the configuration it returns will produce an instance of. And it is the eventual componentClass that is described in the comment to each type method.

These type methods define the range of possible brokers.

BrokerConfiguration

Specifying a broker configuration type selects a concrete broker class. This choice affects the level of service supported by the resulting broker. All broker types are compatible with all adaptor, transport, or marshaler types.

BasicBrokerConfiguration

Supports a basic broker API, use by an application to start, stop, and otherwise control a broker.

StandardBrokerConfiguration

Provides an extended API, which allows an application to use a service registry, and control the list of open selectors.

AdaptorConfiguration

The choice of an adaptor configuration selects a concrete adaptor class. This choice affects the way in which Smalltalk messaging semantics are mapped to the underlying remote messaging layer. Not all adaptor types are compatible with all transport or all marshaler types.

ConnectionlessAdaptor

The adaptor for connectionless protocols, where the same instance of a transport can be used to send messages to several destinations. Compatible with udp and chttp transport types.

ConnectionOrientedAdaptor

The adaptor for connection-oriented protocols, where the transport maintain a listener waiting for connection requests and manages a collection of active connections. Compatible with tcp and iiop transport types.

ObjectGroupAdaptorConfiguration

The adaptor for connectionless one-way protocols. Compatible with bcast and mcast transport types.

TransportConfiguration

The choice of a transport configuration selects a scheme for package handling. A package is the set of bits shipped over the wire. A package has a header, usually transport-specific, that is distinct from the package's payload. The payload, the message the package contains, has a encoding specified by a marshaler. The transport is responsible for managing the conversion of packages into decoded messages and vice versa, for matching incoming replies with previously sent requests, for acting on the information present in a package header, and the like. The primary differentiator of transports is whether they are compatible with a connectionless or a connection-oriented adaptor. Their secondary differentiator is the marshaler or set of marshalers they can work with.

BcastTransportConfiguration

The connectionless UDP transport for broadcast messaging, currently used only with the stst marshaler types.

IIOPTransportConfiguration

The CORBA transport, used only with a cdr marshaler type.

McastTransportConfiguration

The connectionless UDP transport for multicast messaging, currently used only with the stst marshaler type.

TCPTransportConfiguration

The standard, connection-oriented TCP transport, currently used only with the stst marshaler type.

UDPTransportConfiguration

The standard, connectionless UDP transport, currently used only with the stst marshaler type.

MarshalerConfiguration

The choice of a marshaler configuration selects the encoder/decoder that will be used to translate Smalltalk objects to and from a standard, on-the-wire encoding.

STSTMarshalerConfiguration

The marshaler the translates Smalltalk objects to and from the STST encoding used for communication between VisualWorks or ObjectStudio images.

RequestDispatcherConfiguration

The choice of a request dispatcher configuration selects the regimen used to assign an incoming request to the Smalltalk process that will produce a reply.

HighLowRequestDispatcherConfiguration

Forks a new process for each incoming request, but may be configured, depending on characteristics of the request such as its selector, to fork some processes at a higher level than the worker process priority.

PoolRequestDispatcherConfiguration

Puts each incoming request into a SharedQueue with a fixed number of processes as its consumers.

StandardRequestDispatcherConfiguration

Forks a new process, at the worker process priority, for each incoming request. This is the default request dispatcher.

SchedulingPolicyConfiguration

CyclicSchedulingPolicy

A cyclic scheduling policy simply rotates a process queue.

LotterySchedulingPolicy

A lottery scheduling policy assigns each process a number of lottery tickets and conducts a lottery that picks one ticket. The process owning the winning ticket is put to the head of the process queue.

Configuration Parameters

In the following, all the broker and broker component configuration methods are listed under the configuration class that declares them.

Each method parameter, whether a subconfiguration or a parameter, is described as either required or optional. Required parameters must be included in a configuration. They specify some essential, defining characteristic of a broker or broker component, usually the configuration of a broker subcomponent. They usually do not have a

default value. Optional parameters specify some more peripheral characteristic of a broker or broker component. They always have a default value.

Under each parameter, its default values are listed. Because default values are specified in component rather than in configuration classes, there may be more than one default value.

RestartProtocolConfiguration

Components that need to or should respond to system events -- brokers, free-standing objects adaptors, and priority-level schedulers -- inherit from RestartProtocol, and their corresponding configuration class inherits from RestartProtocolConfiguration. Both are abstract classes.

autoRestart: *aBoolean*

Status: Optional

Function: Determines whether a request broker, free-standing adaptor, or priority-level scheduler is automatically restarted after system setup or resume. The options are to always restart (#always), to never restart (#never), or to restart only if the broker or adaptor was running at the time of the preceding system tear down or system pause (#ifQuiescent). If schedulers are used, their autoRestart value should be kept in sync with that of the brokers or adaptors that the scheduler affects.

Default in RestartProtocol: #never

Default in BasicRequestBroker: #ifQuiescent

Default in BasicObjectAdaptor: #ifQuiescent

Default in PriorityLevelScheduler: #ifQuiescent

id: *anObject*

Status: Optional

Function: The identifier under which a broker, free-standing adaptor, or priority-level scheduler is registered in the weak value componentRegistry of class OpentalkSystem. If the user does not create an identifier, one is automatically generated by the defaultComponentIdGenerator defined on the class side of RestartProtocol. The componentRegistry both identifies components that are affected by system events and that it may be useful to programmatically stop or start by id.

Default: An automatically generated symbol consisting of the component class name followed by a millisecond clock value.

BrokerConfiguration

adaptor: *aConfiguration*

Status: Required

Function: The configuration of the broker's adaptor.

Default: None.

requestTimeout: *anInteger*

Status: Optional

Function: The number of milliseconds that a sending broker waits for a reply before raising a timeout exception.

Default in BasicRequestBroker: 60000 milliseconds (1 minute)

AdaptorConfiguration

Note that the processing policy default is set in marshaler configuration classes.

accessPoint: *anIPSocketAddress*

Status: Required, But Optional in a Configuration

Function: The IPSocketAddress at which the adaptor will listen for incoming messages. This is usually set using BrokerConfiguration>>newAt: or BrokerConfiguration>>newAtPort:. But if accessPoint is set in an AdaptorConfiguration, a broker may be created from the enclosing BrokerConfiguration using new.

Default: None

transport: *aConfiguration*

Status: Required

Function: The configuration of the adaptor's transport layer.

Default: None

requestDispatcher: *aConfiguration*

Status: Optional

Function: The configuration of the adaptor's request dispatcher. The are three possible options. A StandardRequestDispatcherConfiguration creates a dispatcher that

forks a new process for each incoming request. A HighLowRequestDispatcherConfiguration creates a dispatcher that forks a new process for each incoming request but forks at one of two priority levels, depending on the characteristics of the request. A PoolRequestDispatcherConfiguration creates a dispatcher that passes incoming requests to a SharedQueue with a fixed number of worker processes as its consumers.

Default in BasicObjectAdaptor: a StandardRequestDispatcherConfiguration

processingPolicy: *aProcessingPolicy*

Status: Optional

Function: A processing policy is used to set up interceptors that perform special actions during the course of message processing.

Default in MarshalerConfiguration: ProcessingPolicy

localityTest: *aBlock*

Status: Optional

Function: Provides a configurable test for locality of ObjRefs. The block takes two arguments; an ObjRef and the adaptor. It answers a Boolean.

ConnectionAdaptorConfiguration

A ConnectionAdaptorConfiguration configures an adaptor used with connection-oriented protocols like TCP/IP. Among its configuration parameters are the three used to make an adaptor work effectively as the conduit for a Web server. In a standard client-server architecture, a server is responsible for serving all connection requests, but on the Web, refusing connection requests is acceptable and useful behavior.

connectingTimeout: *anInteger*

Status: Optional

Function: The purpose of the connecting timeout is to buffer setup delays in a platform's underlying socket implementation. If no connection has been established after the number of milliseconds specified by connectingTimeout the existing socket error is returned.

Default in ConnectionOrientedAdaptor: 1200 milliseconds

connectionTimeout: *anInteger*

Status: Optional

Function: The number of milliseconds that a connection can be idle before being closed.

Default in ConnectionOrientedAdaptor: 1200000 milliseconds (20 minutes)

isBiDirectional: *aBoolean*

Status: Optional

Function: Specifies whether the connection supports 2-way communication.

Default: true

listenerPriority: *anInteger*

Status: Optional

Function: The priority of the process that handles incoming connection requests. The priority of this process should be higher than both the server and the worker process priorities.

Default in ConectionListener: 83

listenerBacklog: *anInteger*

Status: Optional

Function: The connection queue backlog.

Default in ConnectionListener: 1

lowerConnectionLimit: *anInteger*

Status: Optional

Function: One of the three parameters used to control the rate at which incoming connection requests are accepted. The limit at which connection acceps begin to be delayed.

Default in ConnectionOrientedAdaptor: 900

upperConnectionLimit: *anInteger*

Status: Optional

Function: One of the three parameters used to control the rate at which incoming connection requests are accepted. The limit at which connection accepts are refused, and assumed to be higher than lowerConnectionLimit.

Default in ConnectionOrientedAdaptor: 1000

maxAcceptDelay: *anInteger*

Status: Optional

Function: One of the three parameters used to control the rate at which incoming connection requests are accepted. The maximum delay to which delays are progressively increased as the upperConnectionLimit is approached.

Default in ConnectionOrientedAdaptor: 10 milliseconds

soReuseAddr: *aBoolean*

Status: Optional

Function: Turns the SO_REUSEADDR socket option on or off. If the option is enabled, a process may bind to a port number that is already in use. The option should be enabled if you frequently create and destroy brokers on platforms that only tardily release sockets for reuse, or establish multiple broadcast or multicast adaptors on the same port.

Default in ConnectionOrientedAdaptor: false (off)

TransportConfiguration

marshaler: *aConfiguration*

Status: Required

Function: The configuration of a transport's marshaler.

Default: None.

bufferSize: *anInteger*

Status: Optional

Function: Sets the size of the marshaling buffer.

Default in Transport: 1024 bytes

Default in DatagramTransport: 4000 bytes

serverPriority: *anInteger*

Status: Optional

Function: The priority of the process that handles incoming requests by unmarshaling them and dispatching them to a worker process. This priority should be less than the listener priority but higher than both the worker process priority and user scheduling priority.

Default in Transport: 73

DatagramTransportConfiguration

DatagramTransportConfiguration declares an soReuseAddr instance variable at the transport level for use with connectionless protocols.

soReuseAddr: *aBoolean*

Status: Optional

Function: Turns the SO_REUSEADDR socket option on or off. If the option is enabled, a process may bind to a port number that is already in use. The option should be enabled if you frequently create and destroy brokers on platforms that only tardily release sockets for reuse, or establish multiple broadcast or multicast adaptors on the same port.

Default in UDPTransport: false (off)

Default in BcastTransport: true (on)

Default in McastTransport: true (on)

BcastTransportConfiguration

The broadcast configuration code does not follow the standard pattern. It is arguable that netmask is not a parameter in the usual sense. The usual options are either limited or network-directed broadcast.

The default value for netmask is set programmatically in the method BcastTransport groupAddress.

netmask: *aByteArray*

Status: Optional

Function: Determines the scope of the broadcast, making it either limited, subnet-directed, or network-directed.

Default in BcastTransport: #[255 255 255 255] (limited broadcast)

McastTransportConfiguration

The multicast configuration code does not follow the standard pattern. The loopback and ttl values, for example, are better left unset, if a user has no reason to give them a particular value. This is especially true in light of the fact that multicast implementations differ significantly from host to host and OS to OS.

Both loopBack and ttl values are handled in McastTransport>>setOptionsOn:. To turn loopBack on, set its value in an McastConfiguration to true. To constrain ttl, set its value in an McastConfiguration to an 8-bit quantity.

loopBack: *aBoolean*

Status: Optional

Function: Determines whether multicast messages are also received by their sender.

Default in McastTransport: nil (unset)

mcastAddress: *aByteArray*

Status: Optional.

Function: The multicast address joined by a multicast broker.

Default in McastTransport: #[224 5 6 7]

ttl: *anInteger*

Status: Optional

Function: Sets IP_TTL, the upper bound on the number of hops an IP packet may traverse before being discarded.

Default in McastTransport: nil (unset)

MarshalerConfiguration

bufferSize: *anInteger*

Status: Optional

Function: The size in bytes of the marshaling stream.

Default in STSTMarshaler: 1024 bytes

RequestDispatcherConfiguration

workerPriority: *anInteger*

Status: Optional

Function: The priority level of the processes tasked to respond to incoming requests. This priority level should be less than both the server and the listener processes. The worker priority is customarily set to a value greater than the user scheduling priority, but it need not be.

Default in RequestDispatcher: 67

HighLowRequestDispatcherConfiguration

Setting a value for discriminationBlock is technically optional, but the supplied default value is unlikely to be useful in practice.

discriminationBlock: *aBlockClosure*

Status: Optional

Function: A one-parameter block, taking a remote request as its argument, that, if it evaluates to true, causes the request to be forked at highPriority rather than at workerPriority.

Default in HighLowRequestDispatcher: a block that answers true if the message selector matches '*high*'.

highPriority: *anInteger*

Status: Optional

Function: The higher of the two priority levels at which worker processes should be scheduled. This priority level should be lower than server priority (73) but higher than worker priority (67).

Default in HighLowRequestDispatcher: 71

PoolRequestDispatcherConfiguration

The PoolRequestDispatcher should be used when a limit must be placed on the growth of the number of worker processes.

processNumber: *anInteger*

Status: Optional

Function: The number of worker processes that consume from the SharedQueue to which incoming requests are dispatched.

Default in PoolRequestDispatcher: 3

PriorityLevelSchedulerConfiguration

lowerThreshold: *anInteger*

Status: Optional

Function: The size that the quiescent processes list must exceed to trigger scheduling activity.

Default in PriorityLevelScheduler: 1

scheduledPriority: *anInteger*

Status: Optional

Function: The priority level of the quiescent processes list that the scheduler affects.

Default in PriorityLevelScheduler: RequestDispatcher defaultWorkerPriority

schedulingPriority: *anInteger*

Status: Optional

Function: The priority of the process that executes scheduling actions. This priority value should be greater than that of scheduledPriority.

Default in PriorityLevelScheduler: 71

schedulingInterval: *anInteger*

Status: Optional

Function: The interval in milliseconds between invocations of the scheduling action.

Default in PriorityLevelScheduler: 331 milliseconds

schedulingPolicy: *aConfiguration*

Status: Required

Function: A SchedulingPolicy defines the scheduling action, the manner in which the quiescent processes list is reordered.

Default: None.

LotterySchedulingPolicyConfiguration

newMultiplier: *anInteger*

Status: Optional

Function: The factor by which the number of lottery tickets held by hitherto unscheduled processes is multiplied before a lottery ticket is drawn to select one process for relocation to the head of the queue.

Default in LotterySchedulingPolicy: 1

oldMultiplier: *anInteger*

Status: Optional

Function: The factor by which the number of lottery tickets held by previously scheduled processes is multiplied before a lottery ticket is? drawn to select one process for relocation to the head of the queue.

Default in LotterySchedulingPolicy: 1

Network Configuration

An Opentalk broker must know the IP address of its host to function properly. Unfortunately there is no standard, cross-platform way to obtain this information. To derive this information, Opentalk relies on following procedure (implemented in GenericProtocol class method setHost):

1 It obtains the name of its host by sending getHostname.

2 It asks the OS to convert the name to an IP address by sending hostAddressByName:.

This usually works, but can fail depending on the network configuration of the host.

It is becoming more common on Unix platforms (e.g., Mac OS X and some Linux distributions) that the host name resolves to the "localhost" address, 127.0.0.1. This is usually caused by **`/etc/hosts`** mapping the hostname to the localhost address. In this case the Opentalk broker is only able to communicate with brokers on the same host.

It is also possible that the hostAddressByName: call will fail (with an exception) for some reason. In some configurations it may actually invoke DNS to resolve the name, and if the DNS server is not responding the call may block the VM for some time before it fails.

To avoid many of these failures, setting the address explicitly may be the best solution.

Set Host IP

To avoid using the auto-configuration, set the value of the variable GenericProtocol.HostAddress to the appropriate IP address, or GenericProtocol.HostAddresses if the host has multiple addresses. The setter message is hostAddress:.

Note, however, that in order to facilitate transparent migration of images from host to host, this variable is flushed on image startup so that the address can be rediscovered. Therefore the address has to be reset after every image start up.

To set the IP upon start up, create a Subsystem subclass with OpentalkSystem as its prerequisite, and set the variable in its setUp method. For example, create a subclass MyAppSystem, and define these methods:

```
prerequisiteSystems
    ^Array with: OpentalkSystem

setUp
    super setUp.
    GenericProtocol hostAddress: #[ 128 16 16 101 ]
```

This assumes the host has a single network interface. If the host has multiple network interfaces, use hostAddresses instead, which takes an array of interface addresses.

For more information on Subsystems, refer the the Application Developer's Guide.

STST and Firewalls

STST supports pass-by-reference semantics for requests to a broker behind a firewall.

To enable pass-by-reference semantics, the broker needs to advertise its external (firewall) address in the object references that it generates. This provides clients with a valid address to connect to. The client brokers then connect to the firewall, and the firewall forwards those connection requests to the server broker behind it. Of course, the firewall has to be properly configured for port forwarding.

To configure the broker for this type of setup, create the broker using

```
BrokerConfiguration newAt: anAddress
```

The address parameter must be the external (firewall) address through which the broker can be accessed. The same applies to the various instance creation methods on the class side of BasicRequestBroker.

Note that, currently, the port number that is opened on the firewall to forward traffic to an STST broker must be the same as the port number used by the broker on the physical host that it runs on. So if the open port on the firewall is 4242 the broker will also bind to port 4242 on its host machine.

With bidirection enabled, the server can reuse a previously established connection from the client to deliver requests from the server.

Without bidirection enabled, if the server needs to send a request back to client, the client must provide accessible address in the corresponding object reference. If the client is sitting behind a firewall, it will have to be configured similarly as the server, with an external access address and with the firewall configured to forward to the client.

If the address parameter to the broker creation methods is neither a wildcard address (0.0.0.0) nor completely unspecified (via #newAtPort:), the broker treats it as an external (firewall) address. The broker binds to all local interfaces and advertises the external address in all object references that it generates. The broker distinguishes a local address from an external one by obtaining the host name from the operating system and converting it to an IP address using reverse DNS lookup (see the GenericProtocol class method hostAddress).

Bidirectional Support

Bidirectional connection enables Opentalk clients to access a server from behind a firewall.

Without bidirectional support, a server is unable to send requests back to the client unless that client is exposed through the firewall to incoming connections. This is usually blocked by client firewalls.

With bidirectional connections, the server is able to reuse a previously established connection initiated by the client, which is usually allowed by client firewalls, for requests sent from the server to the client, such as callbacks to client objects.

Bidirectional connection can also allow better network resouce management on heavily loaded servers, by eliminating the need for two separate connections per client when there are requests flowing in both directions.

Bidirectional support is controlled by setting the isBiDirectional variable in ConnectionAdaptor:

- isBiDirectional: true enables bidirection support
- isBiDirectional: false sets the connection to single-direction only

Currently only STST provides bidirection support. For any other protocol, an asymmetric connection is created. (Many protocols, such as HTTP, require that anyway.)

It is safe to mix bidirection capable and incapable brokers arbitrarily; they will simply fall back into the asymmetric connection mode. Only two bidirection capable brokers can take advantage of bidirectional connections.

Special Note about Client Configuration

If the client brokers are not explicitly set up with distinct access points, then they will most likely pick up their local host IP, usually from one of the private network ranges. If several clients are in this situation, it may happen that they end up advertising the same address/port for their objects, and the server will not be able to distinguish them. The client broker that is associated with that address (usually the first to connect in bidirectional mode) will receive all requests to that address/port. Without bidirection support, the request attempts from the server will simply fail in these cases, but with bidirection mode they may just happen to work with unpredictable, possibly catastrophic results.

Accordingly, in a network environment supporting bidirectional connections, it is advisable to require the client brokers to be configured with their external firewall address. That will guarantee uniqueness, and ensure that clients can be distinguished on the server side. No holes are needed in the client firewalls, because requests will take advantage of bidirectional connections.

6

Processes, Connections, and Scheduling

An Opentalk request broker runs within the context of the default VisualWorks process scheduling model. This has several implications for the way an Opentalk broker operates. This chapter sets out these implications, and explains both the connection control and the scheduling code shipped with the Opentalk Base.

OS Processes, Threads, and Smalltalk Processes

On conventional operating systems, the VisualWorks object engine runs as a single, OS-level, heavyweight process with one thread of control. The object engine's process scheduler shares that single, OS-level process among several native Smalltalk processes, according to the scheduling system implemented in class ProcessorScheduler and described in the Application Developer's Guide.

Native Smalltalk processes are not, by default, mapped to OS-level lightweight processes or threads.

This architecture has advantages not provided by an architecture that did map native Smalltalk processes to OS-level threads. Because the VisualWorks scheduling regimen is deterministic and reflected in the image, both the object engine and Smalltalk code are also deterministic and therefore comparatively easy to model and to debug. The architecture also makes VisualWorks code more efficient than it would be otherwise, because switching between two native Smalltalk processes is faster than switching between two OS-level threads.

One potential weakness of the architecture arises when the heavyweight Smalltalk process performs I/O operations that may block the object engine until the I/O operation completes. This weakness is addressed by THAPI, discussed in the DLL and C Connect User's Guide. Under THAPI, whenever a native Smalltalk process invokes a potentially blocking I/O operation, a separate OS-level thread is created and passed the information required to make the callout.

Another potential weakness is a consequence of the fact that a single VisualWorks object engine cannot take optimal advantage of a multi-processor machine. A multi-processor machine can, in general, run as many heavyweight processes, without any timesharing, as it has processors. One VisualWorks object engine, since it consumes just one heavyweight process, cannot therefore take full advantage of the presence of two or more processors. Hence, it is common practice — when running VisualWorks on production machines providing a single service — to run, on each machine, a number of images equal to the number of primary processors on the machine. Users of the service can then be routed to the least loaded image that provides it.

Running multiple images on a multi-processor machine is recognized to be a partial solution to this problem. Future improvements, involving engine enhancements that allow multiple copies of the same image to share the same code, and support for concurrent programming, are planned.

However, there are other consequences implied by the VisualWorks process architecture that are particular to the operation of Opentalk request brokers.

Opentalk Subsystem

Opentalk uses the VisualWorks Subsystem framework for responding to certain system events, such as start up and shut down. For general information about subsystems, refer to the VisualWorks Application Developer's Guide.

Opentalk specific handling for these events is defined in OpentalkSystem, a subclass of Subsystem.

Brokers can configure their behavior at image restart as:

- always restart after an image snapshot or shutdown (#always)

- restart only if they had been running prior to the snapshot or shutdown (#ifQuiescent)
- never restart (#never)

The policy is implemented in class RestartProtocol. An application rarely may need to change the default value, #ifQuiescent.

OpentalkSystem also maintains a registry of brokers, giving an application a way to know what brokers are active in the system. Brokers are automatically added to the registry when created. The registry is weak, so brokers can be garbage collected when no longer referenced. To retrieve the registry contents, send a componentRegistry message to OpentalkSystem.

Opentalk-Specific Issues

Broker-specific process issues are a consequence of the interplay between the Opentalk request broker architecture and the VisualWorks process scheduling system.

Any Opentalk request broker runs three critical sets of processes:

- Each running broker maintains a single listener process, at listenerPriority (83), that listens for incoming connection requests. When a connection request is received, a connection is established.
- Each established connection involves a server process, running at serverPriority (73), that listens for messages coming in along the connection.
- Whenever a message is received, the server process spawns a another process, running at workerPriority (67), that services the request, by sending the message specified in the request to its intended, local receiver, a replying to the message's remote sender.

The VisualWorks process scheduling model is based on two principles:

- Higher priority processes are scheduled before lower priority processes.
- Processes at the same priority level are scheduled in the order in which they have been created.

These two claims are, of course, contingent on the assumptions that none of the involved processes include code that yields control, or spawns other processes at a higher priority level than that of the spawning process. But, none of the Opentalk broker code violates these assumptions.

However, these two principles, in conjunction with the process architecture of Opentalk, entail three consequences:

- An Opentalk broker may choke under a flood of connection requests. During a long burst of connections requests, the sustained activity of the connection listener process will disallow the running of the lower priority Opentalk server or worker processes.
- An Opentalk broker may choke under a flood of message traffic. During a long burst of incoming messages, the sustained activity of server processes will disallow the running of the lower priority worker processes.
- An Opentalk broker will always run worker processes in the order in which they were created. As a result, messages that require a long-running-worker process to respond to them, will be executed before short-running ones, if the former were received first. In some cases, this is not optimal behavior.

None of the circumstances mentioned above are frequent or particularly disabling. DST, using the same process architecture as Opentalk, has been used in message-intense enterprise applications for decades, with scant complaint. But each case deserves some additional discussion.

Connection Request Overload

In standard client-server architectures, it is extremely rare for a server's request broker to be overloaded with client connection requests, which consume both time and memory. The communication between clients and servers is well ordered. The connections established between clients and servers are comfortably finite. And, in a standard client-server architecture, it is a server's primary responsibilities is to be there, to never refuse and to happily service every connection request. This is one of the several reasons why the Opentalk listenerPriority is higher than its serverPriority and its workerPriority.

However, the same considerations do not hold for Web servers. On the Web, heavy bursts of connection requests are not uncommon, and when they do occur, it is better to ignore new connection requests than to fail to service those already established.

This is the consideration that stands behind the APIs for setting concurrent connection limits, in ConnectionAdaptorConfiguration and ConnectionOrientedAdaptor.

There are two connection limits, a soft lowerConnectionLimit and a hard upperConnectionLimit. It is assumed that lowerConnectionLimit is less than upperConnectionLimit. A broker event is generated when the number of existing connections exceeds either of these limits. They are #reachingConnectionLimit:with: and #reachedConnectionLimit:with: respectively. Further, when the upperConnectionLimit is passed, the listener process is suspended. Thereafter, as connections are dropped, a #leavingConnectionLimit:with: is generated, until the count falls below the lowerConnectionLimit. Then, a #leftConnectionLimit:with: event is issued, and the connection listener process is resumed.

Both connection limits are configurable. Also, within the range of the two limit boundaries, there is a delay between connection accepts, that progressively grows as the upperConnectionLimit is reached. The growth of the delay is bounded by the configurable parameter maxAcceptDelay.

This facility for controlling the listener process effectively addresses the problem of connection request overload.

Message Request Overload

A broker's server process may be flooded with messages, irrespective of whether the broker is also bombarded with connection requests. When receiving a burst of messages, the server process will consume both space and time as it forks processes, at the lower, workerPriority level, to reply to messages. In very extreme cases, in may take minutes for the burst to end, minutes until worker processes may begin to run and produce replies.

This is a straightforward resource problem, that is not significantly impacted by scheduling regimens. A time-slicing scheduler and a deterministic scheduler like that in VisualWorks will both, in the face of an intense message burst, show a delay in the production of replies to remote requests.

The response to the problem is to add more resource, in particular, more hosts running VisualWorks server images. The messages in an intense burst may then be distributed among several hosts and images using a load balancing scheme.

Adding more resources and load balancing will effectively address the problem of message request overload.

Message Processing Order

In the absence of yields or forks in the code that implements the methods invoked by remote requests, requests sent to an Opentalk broker will be answered in the order in which they were received. This is a simple consequence of two facts

- Processes at the same priority level are scheduled in the order in which they were created.
- By default, an Opentalk server process spawns all worker processes at the same priority level.

Answering requests in the order received is a useful property for any node in any distributed system to have. It is good to know why this is so.

Benefits of Order

In asynchronous systems, message senders do not block after a message is sent to wait for a reply. If a non-blocking client sends three messages to the same server, there is nothing to ensure that they will be received in the same order. Suppose, however, that each message records its originator and the time when the message was sent. A server could, then, by examining the origin and time values, ensure that it processed each message from the same client in the order in which it was sent. It would be critical for the server to do if those messages entailed some change in the server's state, and the client code implicitly relied on the correct ordering of those state changes. This would be even more important if significant state was distributed among several nodes in the network, and the client's messages to the server involved callbacks to the sending client, or further calls out from the server to other nodes, which might in turn make further callbacks to either the server or the client that originated the message train. Distributed execution trains can be arbitrarily

complex. For such reasons, asynchronous distributed systems often go very far out of their way to ensure that well-defined constraints on the order of message delivery are fulfilled.

The issue of message order is far less pressing in the case of synchronous systems. In such systems, all message senders, whether clients or servers, block, and wait for a reply to every request they send, before sending another. If everyone waits for a reply, total ordering of message delivery is ensured.

Note, however, this ordering may be disrupted if any of the methods invoked in the course of sending a message train or replying to a single message yield control or otherwise play with process priorities. If they do, the total ordering of message delivery among the several interacting nodes of a stateful distributed system may be disturbed, with consequent impact on the integrity of the system's state.**Caution:** Think carefully before you issue yields or change process priority in any method invoked by a remote message.

This is why building a server to, by default, reply to messages in the order in which they were received is a good idea. It is one of several reasons why the Opentalk brokers qualify as correctly designed.

Nevertheless, in simple synchronous systems, with few nodes and very short chains of distributed computation, or with servers that are stateless or nearly so, the order in which servers reply to client requests is not a grave issue. A stateless server, or one with no state variables dependent on interaction with multiple clients, need not care about the order in which it replies to a set of messages from several state-independent clients. If the clients block, thus maintaining the ordering of the requests relevant to their state, then a server need not be scrupulous about the order in which replies to client requests. Furthermore, in this case, a time-slicing regimen may outperform one that does not time-slice.

Bi-Modal Message Streams

Consider a burst of remote requests that is bi-model with request to server-side execution time. About half of the messages take the server image a long time, say, 1000 milliseconds, to respond to; and about half take a short time, say, 10 milliseconds. Suppose that the server image always deals with a stream of requests that is bi-modal in this sense. There are about as many very short-running as long-running processes spawned by the server image's request broker.

Also grant that total client-side wait, the sum of the time that clients spend waiting for a reply from a server, is a reasonable measure of performance. For the sake of this example, assume that total-client-side wait can be reduced to an accumulation of server-side execution times, (even if unmarshaling, marshaling, and message transit time would ordinarily be included).

Then, let us narrow the focus to consider a short, four message burst in the entire request stream. Suppose that four different clients have sent messages to the same broker. They arrive at virtually the same time. In response the broker has spawned four worker processes. Suppose their executions times in milliseconds are as follows:

```
1000
1000
10
10
```

Since a VisualWorks server will run these processes from first to last, the total client-side wait, will be:

```
1000 +
(1000 + 1000) +
(1000 + 1000 + 10) +
(1000 + 1000 + 10 + 10) = 7030
```

The first client waits 1000 milliseconds, the second waits 2000 milliseconds, the third waits 2010 milliseconds, and so on.

Then suppose the order of the messages was reversed, as in the following list of execution times:

```
10
10
1000
1000
```

In that case, the total client-side wait would be

```
10 +
(10 + 10) +
(10 + 10 + 1000) +
(10 + 10 + 1000 + 1000) = 3070
```

3070 is far less than 7030. So, from the point of view of the clients, the server would appear more responsive if the requests that it took less time to process were completed before those that took more time.

On the basis of such cases, some have argued that Opentalk brokers should support time-slicing, a scheduling regimen that distributed CPU time among all pending remote requests or worker processes. To see how this might work, suppose again that there are four requests or worker processes on the queue with the following execution times in milliseconds:

1000
1000
10
10

Also suppose that a time-slicing scheduler cycles through these processes, allocating CPU time to them in 10 millisecond increments. Then the total client-side wait has to be accumulated in terms of several passes over the set of processes, and we arrive at the somewhat different calculation shown below:

(10 * 2) + 10 +
(10 * 2) + 10 + 10 +
(10 * 2) + 10 + 10 + (98 *20) + 10) +
(10 * 2) + 10 + 10 + (98 *20) + 10) +10 = 4100

The first 30 milliseconds of execution takes 10 milliseconds off each of the 1000 millisecond requests and finishes the third request. The next 10 millisecond allocation finishes off the fourth request. At that point only the two long running requests remain, each with 990 more milliseconds to go. So, 98 twenty millisecond passes plus another 10 milliseconds will finish off the third request. Then, another 10 milliseconds will finish off the last.

Time-slicing is better than the average of the two examples that did not involve time-slicing:

(7030 + 3070) / 2 = 5050
4100 < 5050

Sadly, these three examples fail to reveal the whole picture. The outcome of a simple cyclic time-slicing regimen are highly sensitive to the characteristics of the request set and to the amount of CPU time allocated per pass by the time-slicer.

Suppose that the requests did not exhibit a bi-modal distribution. Suppose that they all consumed 505 milliseconds of execution time, as follows:

505
505

505
505

Without time-slicing, the total client-side wait is:

505 +
505 + 505 +
505 + 505 + 505 +
505 + 505 + 505 + 505 = 5050

But, if we used a cyclic time-slicer, with a 5 millisecond cycle, the total client-side wait would be:

(100 * 5 * 3) + 5 +
(100 * 5 * 3) + 5 + 5 +
(100 * 5 * 3) + 5 + 5 + 5 +
(100 * 5 * 3) + 5 + 5 + 5 + 5 = 6050

Here, time-slicing underperforms. In general, time-slicing schedulers do not do well unless they are very clever, or happen to be supplied with a set of processes to schedule that exhibits the anticipated variation in execution costs. Note also that the examples above have assumed that there is no overhead involved in the context-switching that frequent time-slicing entails. That is not true. Yet, despite these reservations, and because there are a few circumstances in which time-slicers do well, Opentalk comes with optional scheduling code.

Opentalk Schedulers

Opentalk schedulers are available in parcel Opentalk-Schedulers.

The Opentalk schedulers are implemented using a high priority process that, at a set interval, rearranges the quiescent processes, at a single lower priority. The high-priority process is called the scheduling process. Its priority is called the scheduling priority. The priority of the process queue that it rearranges is called the scheduled priority. It is assumed that the scheduling priority is higher than the scheduled priority. It is also assumed that the scheduled priority is carefully chosen, and one not used for system-level processes.

The default Opentalk process priorities (67, 73, and 83) are all prime numbers, and not used by any other system-level process. Thus, it is a straightforward to select Opentalk's workerPriority (73) as the scheduled priority. Note that if an image is running more than one broker, and both spawn worker processes at the default worker priority, then the worker processes of both brokers will be affected by an Opentalk scheduler that schedules the quiescent process list at priority 73.

The scheduling process is a loop with a delay. Whenever the delay expires the scheduling process invokes its scheduling action. The scheduling action rearranges the quiescent process list at the scheduled priority. The exact nature of that rearrangement is defined by a scheduling policy. Two scheduling policies are provided:

CyclicSchedulingPolicy

This is a simple cyclic or round-robin scheduler of the kind described above. The scheduling action is simply moving the first process on the scheduled priority queue to the end of the queue.

LotterySchedulingPolicy

This is a simple lottery scheduler. At the start of the scheduling action, each quiescent process at the scheduled priority, is assigned a variable number of lottery tickets. Then, a winning ticket is randomly selected, and the process with the winning ticket is put to the head of the process queue. The number of tickets assigned to a process is a function of the number of times the process has been at the head of the queue in the past. Processes assigned more tickets are more likely to win the lottery.

Users may implement lottery schedulers of their own, that assign tickets on the basis of other properties. The lottery model is flexible, and schedulers based upon it are common.

Cautions

The presence of the Opentalk scheduling code answers the arguments of those who feel it would benefit them. However, users are reminded to employ the Opentalk schedulers only under the following conditions:

- The semantics of their distributed application does not depend on faithfully maintaining the order of requests in the order or replies.
- Their servers provably deal with bi-modal or multi-modal request streams.
- They have experimented with several, sample request streams, that typify the actual service loads experienced by their application, to prove that adding time-slicing machinery to a server image's execution overhead entails a decrease in client-side wait, or some other relevant performance measure.

7

Hints for Distributed System Design

With the Opentalk Communication Layer in hand, you will want to start developing multi-image systems. This section aims to help new users of the Opentalk Communication Layer avoid the most common design and implementation mistakes, and to introduce them to the most commonly useful system components and design patterns. In particular, we hope to begin to make you aware of some of the usual design options and design requirements.

We will focus the discussion around a few of the particular issues involved in the construction of systems that communicate, over channels with limited throughput and variable latency. In some cases, we will introduce sample problems, present solutions, and describe the logical components involved in those solutions. In others, we shall simply make observations that we hope are useful. The observations are often in tension, if not opposition: good distributed system design requires balance and judgment. Only experience teaches this; and if experience is the accumulated result of getting oneself out of peril, it is surely complemented by “imperiance,” the art of getting oneself into perils of the interesting, informative, and survivable sort. Test things for yourself.

Readers interested in a more comprehensive treatment are encouraged to peruse the relevant works mentioned in Annotated References, and to examine the bibliographies of those works for useful, supplementary citations.

Shared Objects

Publicly available objects in a distributed environment are shared. In other words, distributed systems, from the standpoint of any client node in the environment, act like multi-threaded systems.

Problem

The fact that several critical objects in a distributed environment are shared resources means that you cannot send two messages in succession to a shared object, and assume it has not received other messages, from other clients, between them. The object may have changed state between your two message sends. You have this problem even if your two messages initiated separate transactions. Another transaction, initiated by another client, may have been interleaved.

Solution

There are two standard solutions to this problem:

- Implement transactions, so that several discrete client messages can be sent within the same transactional context. Once Opentalk supports transactions, this will be a possible. However, it will always be a comparatively expensive solution.
- If you really cannot afford a state change in the recipient between your two messages, combine them into one message. Since VisualWorks uses only one host OS process, and Smalltalk processes are not mapped to host OS threads, and the Smalltalk processes that respond to remote requests are forked at the same priority level, remote messages will usually execute in an acceptably atomic fashion.

Observation

The reasoning of the second solution assumes that you do not play with Smalltalk process priorities in the server-side implementations of the methods invoked by client requests. It is a bad idea to do so, unless you are very sure of what you are doing: the default request brokers depend on the relative priorities of broker listener processes, request servicing processes, and those of the usual, local processes.

Garbage Collection

VisualWorks images collects local garbage, and Opentalk does not yet support distributed garbage collection. Therefore, the existence of an object reference, in image A, to a remote object in image B, does nothing to prevent image B from garbage collecting the object in question.

Problem

In particular, if you send an ordinary instance creation message to a remote class, that returns a reference to it, that instance, in most cases, will be garbage collected, and the reference you have to it will become invalid. If you send a message to the reference after the object has been garbage collected, you will get an OtEInvalidObjectReference error.

Solution

If you want to preserve the validity of remote references to an object, you have several options.

- If the object is a singleton instance of a class, ensure there is a local reference to it in a shared variable or a class instance variable of its defining class.
- Store the remotely created object in a naming service, or some other referenced collection, co-located with the object.
- Create a repository or factory or resource manager that will retain the objects you create remotely.

Note that any object created in these ways must be explicitly destroyed when no longer needed.

Solution Components

The notions of a repository, a factory, and a resource manager overlap. You may view them as portraits of the same ideal component, painted from three perspectives: in terms of one possible implementation, in terms of one possible general access model, or in terms of a desirable set of high-level responsibilities.

Repository

Repositories are a common construct in GemStone development, and they can also be used for storage in memory. A repository is usually a singleton instance of its class. It usually contains and

manages access to only one kind of object. For example, in a nautical architecture application, the singleton instance of HullPlanRepository would contain all the instances of class HullPlan. The repository presents an interface for the creation and destruction of, and keyed or indexed access to, the objects it contains. It will usually also support methods for accessing sets or instances using select, reject, or detect blocks. In a distributed environment, it will be tuned to deliver references, copies, partial copies, and subcollections or iterators, of its contents, to meet performance requirements. In practice, much of the functionality of a repository is a consequence of the fact that it is implemented in GemStone, and thereby partakes of GemStone's security and transaction policies.

Factory

The notion of a factory is part of the OMG COS Lifecycle Service. To create an object remotely, clients take three steps:

1 Obtain a reference to the factory finder at the target creation locale.

2 Obtain a reference to the relevant factory from the factory finder.

3 Send an object creation message to the factory for a factory. Factories are responsible for ensuring that the created object is stored.

In the DST implementation of this service, factory finders are implemented as naming contexts, factories are mapped to classes, and objects created through the interface are referenced by a shared variable that is a set.

The factory interface and its semantics were designed for remote object creation, but something very like it may be used behind the facade of object access as well. For example, when a server must record client state, a client request for an object may be handled on the server as an object creation request: the server creates a client-specific wrapper around the requested object that records client state as needed, and passes a reference to the wrapper back to the client.

Resource Manager

A resource manager is a component that manages resources of some single type. It is responsible for:

- encapsulating its service and its resources in a useful interface
- enforcing security and other access policies, and
- coordinating concurrent access to shared resources.

Time, Synchronous Systems, and Time-outs

The time spent on a RPC is expended in eight places:

- marshaling the request on the client
- sending the request over the network
- unmarshaling the request on the server
- waiting until the server object is free
- servicing the request
- marshaling the reply
- sending the reply back to the client
- unmarshaling the reply and returning it to the requestor.

One portion of the time is spent waiting. It is, therefore, impossible in a distributed environment to know about the state of another processor with certainty. If it does not reply within some time-out limit, then it could be busy, it could have crashed, or the communication link could have failed. You cannot know, unless you assume that the system is synchronous.

Assuming that a system is synchronous amounts to assuming:

- that there is an upper bound on message delay, which is the time spent on sending receiving, marshaling, and unmarshaling,
- that there is an upper bound on clock drift, so that you can measure time-outs and assume shared units of time and shared times within known bounds of accuracy, and
- that there is an upper bound on the time required for a process to execute a step—including the time spent waiting.

It is the assumptions about these several factors that are wrapped together in the request time-outs that most communication frameworks implement, and particularly in the innocuous looking #defaultRequestTimeout of BasicObjectAdaptor.

Observation 1

Opentalk's hard coded value for #defaultRequestTimeout may not be ideal for your system. But we do try hard to get these things right. Think before you change it. If you do change it, do so with an understanding of what a request time-out is intended to be: the amount of time after which a client can reasonably assume that either the communication link has failed or the server has crashed.

Observation 2

When evaluating the performance of a server, in addition to the base service time, look at the request wait time. The former can be measured on the server in isolation, while the latter can only be measured in a multi-image configuration, that at least models, with acceptable fidelity, the request load the server is designed to support.

Reference, Broker, and Communication Errors

Communicating systems can fail in several more ways than stand-alone systems, because they involve both communication channels and remote processes, each of which may become faulty in several ways. Designers must consider and account for at least the most prevalent failure modes.

Formal discussions of distributed systems spend much time on the classification of failure modes, along the spectrum from "crash failures," where one process goes down and other processes can know that it has (the ideal, unattainable good), to "byzantine failures," where a process fails by inconsistently exhibiting behavior calculated to subvert the correct behavior of others (the ideal, improbable evil).

In practice, these classifications are seldom useful. There is one common failure that apprentice designers of distributed applications often experience and often fail to understand, and understanding it is a good introduction to the wider problem of dealing with the failures specific to communicating systems.

Problem

A request broker runs one process at relatively high priority to receive incoming requests. This process, in most designs, spawns several other processes, running at relatively lower priority to handle the incoming requests. This priority structure is as it should be: the request broker's first responsibility is to be available for incoming

requests, and to do its best not to forget or drop a request. However, this priority structure entails that a request broker can be choked: if a broker is barraged with incoming requests, the image it resides in will spend so much its time forking processes to handle requests, that the image never gets any time to run them and produce responses.

This problem usually manifests itself by a run of communication time-outs, on the several clients that are sending requests to the broker in question. Since the server-side broker is choked, the clients do not receive a response in the amount of time usually allowed for one.

In test lash-ups, where a single client is devoted to nothing other than spawning processes that send requests, you may observe failures on the server side as well, occurring when the server does get a chance to respond. In this case, the client is so busy sending requests that replies do not get through. This results in a server-side communication failure. These failures are dependent on the client-side process priority structure.

Solutions

Scenarios like these do not entail that there is something wrong with the server-side request broker or the request broker design. The problem is in the application design, the deployment design, or the hardware specification. There are at least seven standard solutions, several of which can and should be executed in parallel.

- Get a server-side machine with more horsepower. This always helps. It usually solves the problem outright. In some cases it may not be sufficient to compensate for gross faults in the software or deployment design.
- Create a client-side handler that retries the request in response to a time-out exception. You should do this in all cases, preferably once, at a high enough level in your call tree to ensure that you only need to do it once. Alternatively, wrap the handler in a façade to or wrapper around all remote calls. This is a complete solution in those few cases where the client can afford to wait through several retries and you cannot or do not wish to add more brokers and images to your deployment design.
- Reduce the number of messages that clients must send to do their work. Choked request brokers are often the product of nothing more than chatty client APIs, which send ten messages where one, with several more arguments, would serve. You should check for this possibility in all cases.

- Add additional hosts running server images (or additional server images to a multi-processor machines) and hard code a distribution of the clients among the additional hosts. This will work, but is neither pretty nor amenable to straightforward maintenance.
- Refine the fourth solution by creating a front-line broker. Let one request broker serve as the principle point of contact for all clients. Let it redirect requests, according to some request distribution or load-balancing scheme, to the several back-line server images. Set it up so that the several server images respond directly to the client rather than through the front-line broker. This approach may help, but will not always scale. There is still one entity, the front-line request broker, through which every request must pass.
- Refine the fourth solution by adding a service broker. Let all clients reacquire, from a service broker, the server that the client needs to communicate with at the start of each message send or each major connected group of sends. Implement the service broker so that it enforces a request distribution or load-balancing scheme. Note that in this approach, if you do not ensure that clients reacquire references to their service providers at appropriate intervals, you will subvert any distribution or balancing scheme implemented by the service broker. The other side of the coin is that this approach has slightly better scaling properties than the fifth, when the service broker is addressed by clients only at the beginning of long groups of message sends.
- Use multicast rather than unicast. Send requests to a multicast group and arrange for the servers in the group to negotiate, among themselves, who will respond to a given client request.

Solution Components

Service Brokers

A service broker is a component that answers a reference to a server object when requested for a service provider under a service name. The server object may be a resource manger, a compute service, or any other object that provides a service. Service brokers may, but need not, be request distributors. Sophisticated service brokers may implement several, dynamically alterable brokerage policies, for several service types. It is better to put service brokers in a shared naming service than several entries for providers of the same service.

Request Distributors and Load Balancers

There is a sharp distinction between a request distributor and a load balancer. Both may maintain a working collection of the available servers, but they differ in other respects.

A request distributor hands out requests like a dealer distributes cards: it treats the working collection of available servers as a cycle. It is a good choice in cases where most requests to a service type take nearly the same amount of service time.

A load balancer either polls the servers for some load measure, or expects to receive periodic updates regarding the measure's value from each server. In the former case, the balancer must hold the defining list of currently available servers. In the latter, all the servers must know and report to the load balancer. In either case, the load balancer treats the list of available servers as a sorted collection, sorted on the load measure, and it assigns an incoming request to the foremost (least busy) member of the collection. Load balancing is the required choice in cases where requests to the same service type have high variance in service time.

Load balancing is more expensive than request distribution because it requires communication and coordination between the balancer and the service providers it regulates. Therefore, if a service has high variance in service time as a product of possessing several sub-services, it is useful to partition the service. Request distribution, among instances of the several subtypes of a partitioned service, can be more efficient than load balancing, among the several instances of a non-partitioned one.

Observation

Choking the request broker is an instance of a more general issue. Any request broker architecture will issue several exceptions. It defers the handling of these exception to a higher layer, as being beyond its responsibility, and suitable for selective or discriminatory handling. The higher layer is, as a consequence, responsible for providing the appropriate set of handlers. You cannot design a distributed application correctly without knowing the list of exceptions raised by the underlying communication layer, and assuming knowledgeable responsibility for their treatment.

Scalability and Single Points of Failure

Distributed systems are attractive because they promise scalability. As demand for a service increases, you may, ideally, simply add more servers that provide it.

Observation 1

Even if you have a putatively scalable collection of several servers, among which you distribute or balance service requests, clients using unicast will usually require a single, well-known point of access, that does the distributing or balancing. Because all client requests go through that single point, its optimization is critical.

Observation 2

The single access point is also a single point of failure. If you cannot afford failure, you will need to make the access point fault-tolerant in one of two ways. Either:

- implement a primary-backup framework, or
- implement the service as a group of service providers, and use multicast rather than unicast.

Observation 3

Adding service providers buys scalability only if you do not need to coordinate their state. If there are state coordination costs, adding service providers will usually increase those costs exponentially, and they will eventually swamp the linear gains accrued by service provider addition.

Remote Message Number

Remote messages take more time than local ones, by three to seven or more orders of magnitude. Though a distributed system will hopefully be transparent enough so that an application user or a component developer can ignore whether the messages he sends are going local or remote, the developer of the underlying application distribution layer cannot. In other words, even though Opentalk is designed so that you are not required to know whether an object is local, this does not imply that you always may.

In practice, concern about the comparative cost of local and remote messages often devolves into concern about the number of remote messages sent to achieve some end.

The cost of a remote message is usually a summation of:

- the base cost of sending a remote message,
- a cost factor derived solely from the message length, and
- the marshaling costs specific to the object types linearized during marshaling.

Flattening a complex, cross-referenced object tree usually takes more time than flattening a long array of integers, even if the two eventually consume the same amount of space in a message body.

Observation

You often do not have control over what you have to marshal, but if you can send one message with four arguments instead of two messages with two, you save one payment of the base cost. This is why distributed system designers like shared objects possessing sparse protocols, and supporting method implementations that do a lot for each message sent. Reducing the number of remote messages you send to get a job done is one of the easiest ways to maximize throughput, and throughput is the interesting measure, not bandwidth.

Variable Latency of Remote Messages

A remote message, in addition to taking more time than a local one, takes an amount of time that varies more widely.

Observation

When measuring the time consumed by a remote message, be aware of the fact that you are measuring one point in a range. At some point, you will become concerned with the factors that push the value to one end of the range rather than the other.

Remote Object Representation

The number of remote messages sent to achieve some end is directly affected by whether you elect to locally represent a remote object by a reference, a copy, or some species of partial copy. This

choice is so consequential that no generally useful protocol can be called complete without facilities for pass mode control—the ability to selectively pass an object by reference or by value.

There are several ways of representing a remote object on a client, and it is useful to know the options and understand something about their trade-offs. However, you will need to profile on your own, to arrive at practically useful values for the comparative costs.

Using a Direct Reference to an Application Object

Representing an object by a reference—an instance of class RemoteObject in Opentalk that is a direct reference to an ordinary application object—is good choice when:

- you do not expect to send very many messages to it, and
- you are not interested in controlling access to the object at the level of the remote representation.

You do not want to engage in intense and frequent conversation with a reference because every message sent to it involves network traffic. It is like having an intimate friend that lives on the other side of the country: at some point you notice the phone bill. The flip side is that exporting a reference is usually cheaper than sending a copy, because a copy of an object of average size is larger than a reference. This is another way of saying that even though the phone bills are bad, they are often better than the airline charges.

You need to be uninterested in access control to pass application objects by reference, as a consequence of the fact that the reference may be passed to more than one client. Exporting a direct reference to a name space, for example, allows others to change the name space and all of its contents without any security constraints or concurrency control, beyond those that may be implemented at the level of the request broker. If the broker provides a security service, it can prohibit modification by suspect parties, but does nothing to prohibit contentious modification by trusted ones without additional machinery.

Using a Direct Reference to a Service Provider

A reference to a service provider, such as a resource manager that handles resources of the target type does not involve reservations about access control. A resource manager, or a well-designed and complete service, by definition, handles security and concurrency on its own.

Using a Copy or Replicate

Using a copy is preferred when:

- you expect to send many messages to the object,
- the object is not so large that copying has noticeable performance costs, and
- you can accept the inequality between the copy and the original.

The simplest example of a useful copy is a source code file, checked out of a source code repository. It is an object that:

- you expect to modify heavily,
- is usually of manageable size, and
- you agree to pay the costs associated with either:
    - file locks,
    - merges,
    - the communication costs associated with frequent or intermittent state update, or
    - aborted transactions.

If all you had were references and copies, copies of large objects that you intended to send frequent messages to would still be problematic because of their transfer costs.

Using a Faulting Proxy or Stub

A faulting proxy or stub is a proxy that replaces itself with a copy of the object it refers to when it is first sent a message by application code, and then redispatches the method to the copy. Subsequent messages go to the copy directly.

Faulting proxies are used to reduce the immediate costs of copying a large object tree. At selected points in the object tree, subtrees are passed as faulting copies. The full cost of the copy is paid in installments, rather than at once, and only paid when needed.

The notion of a faulting proxy is elegant, but the task of tuning a copying specification so that faulting proxies are placed at the right level to ensure good overall performance is often tedious and lengthy.

Using a Reference to a Server Mask

We use the term “mask” to refer to a wrapper around an object or reference. A wrapper is understood to be comparatively lightweight: it wraps a single object or reference, unlike a resource manager or repository, which wraps a set of objects of the same type.

A server mask is co-located with the object it masks on the server. The mask may handle various responsibilities. Using a server mask is recommended under several circumstances; the following are the most common:

- When the server needs to record client state during the interaction, a client-specific mask around the target object can be tailored to do this.
- When the target object is represented in an unsuitable or inefficient manner, a temporary mask can be created on the server. The mask will contain a representation that is more effective for handling client queries, and maps between that representation and the one in use on the server. A mask of this sort need not be client-specific.
- When the interface of the target object is unsuitable, it can be wrapped in a simple translating mask. This is one of the easiest ways to prohibit the remote invocation of selected methods, in the absence of other forms of exported interface control.

Masks have creation, storage, management, and design costs. They can be used around both ordinary application objects as well as service providers. They can be designed to handle simple security and concurrency issues.

Using a Client Mask Around a Reference

You can just as easily create the mask on the client side. Using a client mask is encouraged:

- when the client possess the information that determines the optimal interface or representation, or some other aspect of the wrapping policy, or
- it is the client, rather than the server, that has the spare cycles required for mask creation.

The client mask may contain instance variables that cache the results of queries sent to the reference they contain, when those results are known to be sufficiently stable.

Using a Shadow With a Direct Object Reference

We use the term “shadow” for a wrapper created by the server, passed by value to the client, and containing a reference to an object on the server. Because a shadow is passed by value, it must have equivalent implementations on both the server and the client side.

A shadow can implement any of the masking responsibilities mentioned previously. A shadow may also be used as an alternative to a copy containing faulting proxies, because:

- it may contain instance variables preloaded with frequently accessed values, and
- it may also contain empty instance variables, with lazily initializing accessing methods, that retrieve a value from the server on first call.

Thus, shadows may be designed to act like partial copies containing faulting proxies. Tuning a shadow’s instance variables and accessing methods is logically equivalent to tuning a replication specification.

Using a Shadow with a Reference to an Object Manager

You may use a shadow with an object identifier and a reference to an object manager when the manager provides security and concurrency control, and these are required. The identifier is assigned by and meaningful to the resource manager that produces the shadow. The manager uses the identifier to dispatch messages sent to a remote shadow, and dispatched to it, on to the object the shadow represents.

Using Both Client and Server Masks

In some cases, you will use both client and server masks. The client mask will contain a reference to the server mask. This makes sense when either:

- the relevant information needed to set masking policies is present on both the client and the server sides, or
- it makes sense to distribute the cost of creating masks for performance reasons.

Remote Object Number

Some performance problems reduce to questions of scope. The fewer remote objects you have to represent locally, the lower your overall transfer and representation creation costs.

Observation

The impact of your choice of remote representation may be as nothing, when weighed against the impact of the number of remote objects you attempt to represent.

Remote Object Alteration

The previous sections examined the several ways in which a remote object may be represented, and the virtue of requiring few of them. It is an incomplete overview, because it does not exhaust the ways in which a remote object can be changed. At the highest level, there are only four ways to do this.

Send it a Message

You can change a remote object by sending a message, which immediately entails a remote state change, to some suitable local representation.

You may also send a direct message in a transactional context. This may be a distinct enough kind of message to warrant a high-level category of its own.

Replace It

You can also change a remote object by changing a copy of it and then replacing its remote representation with the local copy, passed by value, back to the site of the original, usually to a resource manager that will effect the replacement.

Ship Over the Modifying Code

You may, given the supporting infrastructure and sufficient homogeneity of environment, ship modifying code over to the site at which the object resides, where the server executes it for you. This reduces to shipping, in some form, a block: it is the client that supplies the implementation of the mutating method. The interesting issue is where the compilation occurs.

Create an Agent Which Copies Itself Over and Does the Work

This option only makes sense if you have the supporting infrastructure and are not particularly concerned about the time at and the order in which target objects are altered. Agents do not have much to offer over code shipping unless either:

- they are copying themselves to multiple sites,
- they perform the same job at each site, or perform a discriminated agenda of several actions at many sites, or,
- in the context where several agents are competing for computational resources, you sincerely care little about the timing and sequencing of the changes they effect.

Shipping an agent is nearly the ultimate in asynchrony.

Agents may not be distinct enough from code shipping to warrant a high-level category of their own.

Replication Rate and Replication Delay

When designers of distributed system are not facing throughput issues, they are confronting the tight tolerances present in real-time systems or entailed by digital video and voice data. In such cases, there may be severe constraints both upon the rate at which data must be generated at one location or reproduced at another, and upon the allowable delay between generation time and reproduction time.

A sample rate restriction is the requirement that video data must be generated and reproduced at a minimum of 16 frames per second to avoid the subjective impression of image “jumpiness.” A sample delay restriction is that the reproduction of sound data must exhibit a delay of less than 100 milliseconds, between the termination of a conversational inquiry and the reproduction of an immediate response, to forestall the subjective sense of noticeable hesitation.

Observation

These are the kinds of problems you address by creating a protocol tailored to handle them. You may expect that the most significant part, of any complete and generally useful solution, will be implemented at levels below the transport layer.

Initial Reference Acquisition

The distributed system designer has to address problems and components not present in stand-alone systems. The most obvious is the problem of initial reference acquisition: you cannot talk to any remote object, or get references to other remote objects, unless you have a reference to one of them. When you design a distributed system, you cannot ignore this issue.

Problem

Let us assume that the only initial reference, that all clients need to obtain at start up, is the reference to a single service broker.

Solutions

The following are the obvious possible solutions:

Generate a reference to the service broker

Use unicast. Have each client programmatically generate a reference to the service broker at startup. You can only generate a reference programmatically if you know the address of the broker, and it has a well-known, constant object identifier. Each client has to know the broker's location and address. You will have to alter the machinery for exporting object references, to ensure that the service broker always gets the same object identifier: this solution involves fundamental alteration of the communication layer

Generate a reference to the request broker

As a sub-variant of the previous approach, you can install the service broker as a service published at the level of the request broker. Most request brokers already have a well-known constant object identifier. Clients programmatically generate a reference to the request broker, and then ask the request broker for the service broker under its service name. Clients must know the service name, and still need to know the location of the broker. The latter can be hard-coded, or placed in a file that is read at startup. Neither solution is pretty or easy to maintain. This solution does not involve meddling in the implementation of the communication layer.

Use a naming service

Have clients resolve the name of the service broker, in the naming service to obtain a reference to it. This allows you to shift the location of the image containing the service broker without undue maintenance costs, and this is a plus. But you still have a single point of failure: the naming service. And clients still need to get a reference to it, which means that they need to know its location. They also need to know the name under which the service broker is found in the naming service rather than the name under which it is listed as a broker-level service, as in the previous solution.

Use multicast

Set up the service broker(s) as a multicast group. Clients send a multicast message to the group to gain access to the service broker. Clients only need to know the multicast group identifier. After getting a reference to the particular service provider they require, from the service broker, clients may continue using unicast. The maintenance costs here are minimal.

Observation

Multicast is the right protocol for gaining initial access to shared resources; unicast is not.

Encapsulation and Transparency

All of the differences between distributed and stand-alone systems so far mentioned or implied are differences that the distributed system designer would like to hide from both application-level component developers and application users, so that the overall design scores high on encapsulation and on various dimensions of transparency.

Transparency requirements are diverse and several; many of the works mentioned in Annotated References describe and discuss them.

Problem

The species of transparency that most immediately affects code developers is “access transparency.” This is the requirement that objects be accessed in the same way irrespective of their location. Its corollary is that application developers should not need to know whether an object is local or remote.

This problem is accentuated by the fact that messages to remote objects may deliver special exceptions having to do with reference, broker, or communication failure. The handlers that address them should not be the concern of application-level developers.

Some communication patterns raise special problems of their own. Promises may diminish the impact of asynchronous RPCs on code structure, but do not, in the absence of some encapsulation strategy, hide that fact that you are using asynchronous RPCs in some cases and local calls in others.

No Single Solution

Access transparency can only be achieved by interposing a layer—let us call it “the distribution layer”—between the communication layer and the level at which the application developer works. This layer is the primary responsibility of the distributed system designer.

The implementation of this layer can be facilitated by off-the-shelf components like request distributors, proxies, masks, and shadows, or by off-the-shelf services that support transactions, concurrency control, shared clocks, and the like.

There are several frameworks that claim to provide a complete solution to the designer of this layer. Sadly, applications may have requirements that are not addressed by a specific “solution”: video and voice data have special constraints; cooperative work applications and shared database front-ends have very different communication patterns. In short, there may be no existing recipe, describing the lineaments of a solution, that will work best in support of both your application and the degree of transparency it requires. You will have to research, experiment, and work it out for yourself

Observation 1

You cannot assume that because a protocol framework is designed to provide immediate support for access transparency—as the Opentalk framework does—that you can afford to ignore location when you address the distribution layer.

Observation 2

You cannot assume that, because a distributed application toolkit provides, at some point in time, a given set of off-the-shelf components, that those are the components you should use to solve your problem.

Observation 3

An old study showed that, independent of the implementation language used, and number of years of prior programming experience, developers were wrong 50% of the time about where the time went in the execution of sample blocks of code. The study was conducted assuming a stand-alone system.

If you plan to design and implement a distribution layer, use the profiler.

A

Opentalk, Distributed Smalltalk, and I3

Users of DST will notice similarities between Opentalk and the Implicit Invocation Interface (I3) present in Distributed Smalltalk (DST), and be concerned about the relationship between these two products.

Opentalk is *not* intended to be a long-term replacement for I3.

I3 is an alternative marshaler, built into the DST marshaling framework. It uses the IIOP protocol, and allows DST users working in a homogeneous environment to prototype a DST application without writing IDL. Since such prototypes will often be ported to heterogeneous environments, DST uses the standard DST marshaler if an applicable IDL interface declaration is present, even when I3 is turned on. This allows a DST developer, who is porting an I3 prototype to a heterogeneous environment, to test IDL code incrementally in a homogeneous one.

In the long term, IIOP and DST will be reimplemented under the Opentalk Base. At that time, the motivations for a DST facility like I3 will still exist. I3 will remain, but will be reimplemented to accord with the then current DST marshaling framework.

B

Annotated References

The following collection of annotated references is organized by topic. The topics are arranged alphabetically. The annotations express the evaluations of the authors of this manual.

Communication Protocols

Pete Loshin. *TCP/IP Clearly Explained*. Third Edition. Morgan Kaufmann, San Diego, California, 1997. ISBN 0-12-455826-7.

This is perhaps the most accessible introductory text on TCP/IP and UDP. It is an excellent base upon which to approach more complete treatments.

W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0-201-63346-9.

This is a voluminous work. It has no excuse for omitting anything, and does not. It is the best of its several rivals.

Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, Reading, Massachusetts, 1995. ISBN 0-201-63354-X.

This is the implementation-level companion of the previously mentioned work mentioned. It has the same virtues.

Computer Networks

Douglas E. Comer. *Computer Networks and Internets*. Second Edition. Prentice Hall, Upper Saddle River, New Jersey, 1999. ISBN 0-13-083617-6.

This is a standard text.

Andrew S. Tannenbaum. *Computer Networks*. Third Edition. Prentice Hall PTR, Upper Saddle River, New Jersey, 1996. ISBN 0-13-349945-6.

This is another standard text. It gives more space than does Comer to ATM, and is arguably superior in other respects as well.

Distributed Agents

Jacques Farber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, Harlow, England, 1999. ISBN 0-201-36048-9.

This is an available, complete discussion of the area. It now has no serious current competitors as a general survey that begins at an accessible level. This evaluation is likely to be overturned by forthcoming publications.

Distributed Algorithms

Valmir C. Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, Cambridge, Massachusetts, 1996. ISBN 0-262-02412-8.

This work is more narrowly focused on specific problem areas, and less compendious than Lynch.

Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996. ISBN 1-55860-348-4.

This is the current, standard collection, and superior to the available contenders.

Distributed Systems

Prabhat K. Andleigh and Michael R. Gretzinger. *Distributed Object-Oriented Data-Systems Design*. PRT Prentice Hall, Englewood Cliffs, New Jersey, 1992. ISBN 0-13-174913-7.

This is an older and somewhat dated text. It is nevertheless a useful treatment of distributed system design, as it was understood before the full advent of the current object models, which obscured the importance of some earlier work.

George Couloris, Jean Dollimore, and Tim Kindeberg. *Distributed Systems: Concepts and Design*. Third Edition. Addison-Wesley, Harlow, England, 2000. ISBN 0-201-61918-0.

This is an excellent and highly recommended text.

Sape Mullender, editor. *Distributed Systems*. Second Edition. Addisson-Wesley, Wokingham, England, 1993. ISBN 0-201-62427-3.
Andrew S. Tannenbaum. Computer Networks. Third Edition. Prentice Hall PTR, Upper Saddle River, New Jersey, 1996. ISBN 0-13-349945-6.

This text is weaker than Coloris, Dollimore, and Kindberg on multicast, and suffers from being a collection of papers rather than a single work. It is still excellent, and highly recommended.

CORBA and Smalltalk ORBs

Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns*. John Wiley and Sons, Inc., New York, New York, 1997. ISBN 0-471-15882-8.

This is a collection and description of a few of the common design patterns present in networked rather than distributed systems. It is spotty: you are likely to find no more than twenty pages of immediate practical use.

Ron Ben-Natan. *CORBA: A Guide to the Common Request Broker Architecture*. McGraw-Hill, New York, New York, 1995. ISBN 0-07-005427-4.

This is an older work about a fast-changing specification, but it is one of the few works on the CORBA architecture to devote substantial attention to Distributed Smalltalk.

Terry Montlick. *The Distributed Smalltalk Survival Guide*. Cambridge University Press, Cambridge, United Kingdom, 1999. ISBN 0-521-64552-2.

This book's observations about the comparative merits of the several available distributed Smalltalk implementations are already dated, but it is the only available, recent survey of its field.

Group Multicast

Kenneth P. Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994. ISBN 0-8186-5342-6

This book is mandatory reading for anyone who would rather not reinvent the wheel in multicast group support.

Thomas A. Maufer. *Deploying IP Multicast in the Enterprise*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1998. ISBN 0-13-897687-2.

This book addresses the qualities of several multicast implementations, at the levels below the transport layer, and provides information not readily available elsewhere.

Peer-To-Peer

Andy Oram, ed. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly and Associates, Inc., Sebastopol, California, 2001. ISBN 0-596-00110-X.

This is the best current overview of peer-to-peer technology.

Performance Analysis

Raj Jain. *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc., New York, New York, 1991. ISBN 0-471-50336-3.

This is an excellent and comprehensive text on one of the most important aspects of distributed system evaluation and design.

Index

P

R

S