OrbixTalk Programmer's Guide

IONA Technologies PLC
November 2000

M 2 5 1 5

Contents

Part III Managing OrbixTalk

Preface

OrbixTalk provides a reliable multicast messaging system that supports an implementation of the Common Object Request Broker Architecture (CORBA) Event Service defined by the Object Management Group (OMG). OrbixTalk also provides a MessageStore to add persistent storage, on-demand playback and guaranteed delivery of messages.

Any CORBA application can use the IIOP protocol to connect to the OrbixTalk Gateway in order to send multicast messages, or *events*. Any number of applications can supply these events, and are called *suppliers*. Any number of applications can receive these events, and are called *consumers*. Neither supplier nor consumer need be aware of each other's existence.

Audience

This guide is aimed at programmers who are familiar with C++ programming and basic Orbix programming, as explained in the *Orbix Programmer's Guide C++ Edition*. This guide provides information about writing user applications that use the CORBA Event Service. You can also use the OrbixTalk Application Programming Interface (API) directly.

Orbix documentation is periodically updated. New versions between releases are available at this site:

`http://www.iona.com/docs/orbix/orbix33.html`

If you need assistance with Orbix or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `doc-feedback@iona.com`.

Roadmap

This guide is organized as follows:

Part I Introduction to OrbixTalk

Part I introduces OrbixTalk, the Reliable Multicast Protocol and the OrbixTalk MessageStore.

Part II Developing OrbixTalk Applications

Part II provides the following information:

- How OrbixTalk works.
- How to write applications using the Event Service.
- How to use the IIOP Gateway so that any IIOP-compliant CORBA application can send multicast messages.
- How to use the C++ events library so that C++ applications can use multicast functionality directly.

Part III Managing OrbixTalk

Part III provides information about:

- Useful tools.
- How to build and run OrbixTalk applications.
- Configuration parameters and how they affect each other.
- A list of system exceptions.
- Daemons.
- Troubleshooting.

Part VI Appendices

This manual includes the following appendices:

- Appendix A, "Configuration Parameters" describes the parameters used to configure OrbixTalk.
- Appendix C, "CORBA Event Service: IDL Interfaces" lists the IDL interfaces used by the CORBA Event Service.
- Appendix D, "Using the OrbixTalk API Directly" describes how to develop applications with the OrbixTalk API. It also shows you how to write applications that include the OrbixTalk MessageStore.

  This appendix develops a demonstration program illustrating how OrbixTalk can be used to implement an auctioneer and bidders in an auction scenario.
- Appendix E, "OrbixTalk Class Reference" provides a reference to the classes used in the Orbixtalk API.

Document Conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `cd /users/`*`your_name`*

This guide may use the following keying conventions:

< >	Some command examples may use angle brackets to represent variable values you must supply (this is an older convention).
... ⋮	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Part I

Introduction to OrbixTalk

1

Introduction to OrbixTalk

This chapter introduces OrbixTalk—a decoupled, asynchronous messaging system based on a multicast transport service.

Overview

Traditional systems of inter-object communication have focused on a point-to-point approach with one object communicating with one other object. In many situations, however, there is a requirement for a one-to-many or many-to-many form of communication between objects.

IONA has implemented a messaging system known as OrbixTalk that provides a one-to-many or many-to-many form of communication. OrbixTalk enables objects and applications, running on different hosts within a subnet, to share information. OrbixTalk is a *decoupled, asynchronous* messaging system based on a *multicast transport service*.

A messaging system is said to be *decoupled* when the application sending a message has no information about the objects receiving its message. This enables an application to send messages to a group whose members can be unspecified. In OrbixTalk, the applications sending messages and the applications receiving messages do not require any information about each other. This enables the members of a group to change dynamically without affecting the application sending the message; for example, a television channel does not need to know which televisions are switched on before broadcasting a program and televisions can be switched on and off without affecting the actual program signal.

OrbixTalk provides a reliable *multicast transport service*. The multicast transport service enables an application to send a single message to a group of objects therefore reducing the load on network resources. A messaging system based on a multicast transport service is efficient and easily scalable. For more information about the multicast transport service, refer to Chapter 2 "The OrbixTalk Reliable Multicast Protocol" on page 9.

This implementation of OrbixTalk is based on the Event Service defined by the Object Management Group (OMG), which extends the core Common Object Request Broker Architecture (CORBA) standard. The CORBA Event Service specifies how applications that require decoupled communication can be built. This guide discusses how to build applications with the Event Service in Part II Developing OrbixTalk Applications, "Part II Developing OrbixTalk Applications".

OrbixTalk also provides a MessageStore to add persistent storage, on-demand playback and guaranteed delivery of messages. Applications send messages to the OrbixTalk MessageStore which stores the messages in a database and forwards them to applications waiting to receive these messages.

Using OrbixTalk

The following examples show how OrbixTalk can be used in different situations.

Scenario 1: A Stock Price Reporting System

- A ticker tape sends stock price information to the Stock Price Reporting System; for example, Reuters.
- Stock price information is sent to all services that have registered interest; for example, stock brokers, the Wall Street Journal, the Financial Times, Web reporting tools.

This scenario does not require persistence as previous stock prices are not required.

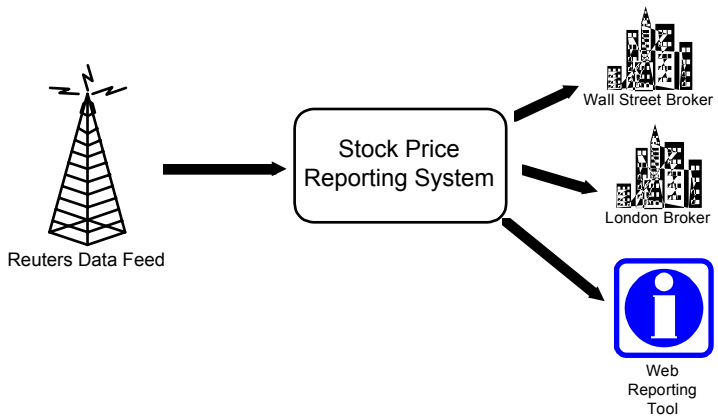

Figure 1.1: *Stock Price Reporting System*

Scenario 2: Travel Agent

- Travel Agent receives updated schedules from the airlines at regular intervals.
- Tourist receives latest information about specific flights from the Travel Agent.

There is a requirement for a persistent store of information so that travel agents can obtain information about flights past and present.

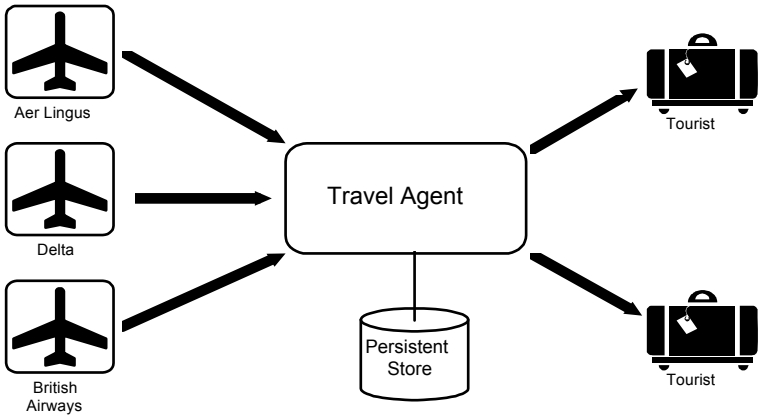

Figure 1.2: *A Travel Agent*

Writing Applications Using OrbixTalk

You can access the functionality of the OrbixTalk multicast messaging system by writing user applications that use the CORBA Event Service. You can also write user applications that use the OrbixTalk Application Programming Interface (API) directly, however, the Event Service approach is preferred.

Figure 1.3 shows the overall architecture of OrbixTalk including applications using the OrbixTalk API directly, applications that use the CORBA Event Service, the OrbixTalk API and the OrbixTalk multicast transport service.

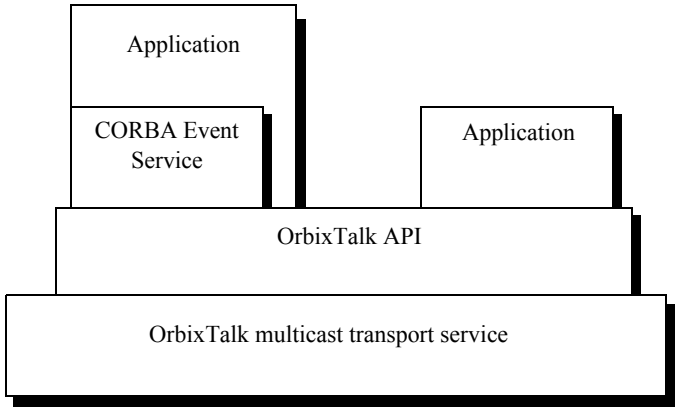

Figure 1.3: *Overall Architecture of OrbixTalk*

Applications that Use the CORBA Event Service

The CORBA Event Service, defined by the Object Management Group (OMG), specifies how applications that require decoupled communication semantics can be built. You create applications based on the concept of *suppliers*, *consumers*, and an *event channel*. Suppliers and consumers can implement push or pull semantics.

Applications that use the CORBA Event Service 'plug in' to the OrbixTalk API to access the functionality of the multicast messaging system. The OrbixTalk MessageStore is also available to provide persistence and playback of events. For more information, refer to "Part II Developing OrbixTalk Applications".

Applications that use the OrbixTalk API Directly

OrbixTalk enables you to access the functionality of the multicast messaging system using the OrbixTalk API. You create applications based on the concept of *talkers* and *listeners* communicating on a *topic*. For more information, refer to Chapter D "Using the OrbixTalk API Directly" on page 219. However, the Event Service approach is preferred and should normally be used.

2

The OrbixTalk Reliable Multicast Protocol

This chapter introduces the Reliable Multicast Protocol used by the OrbixTalk messaging system.

Overview

There are three distinct ways to send messages over a network:

- Unicast

  A message sent from one host specifies the address of a single destination host.

- Broadcast

  Messages are sent from a single host to all other hosts in the network.

- Multicast

  A single message is sent from a host to a set of hosts that belong to a specified multicast group.

OrbixTalk provides a multicast transport service that can be accessed by applications and objects on the network. A multicast transport service enables a single host to send data to many destinations using a single call on the transport service.

A set of hosts receiving messages on a particular IP multicast address is called a multicast group. A multicast group can span multiple networks. Hosts can join and leave multicast groups at any time. Adding hosts to a multicast group does not affect the messages sent on the network—a single message is sent regardless of the number of hosts in the multicast group. In this way, a multicast transport service reduces the load on network resources and is easily scalable.

User Datagram Protocol and Reliable Multicast Protocol

OrbixTalk uses the User Datagram Protocol (UDP) based IP multicast to share information between applications. By itself, this mechanism does not inform applications when information has been lost or arrives out of sequence. To provide reliability, OrbixTalk uses the OrbixTalk Reliable Multicast Protocol (`otrmp`). The OrbixTalk Reliable Multicast Protocol ensures that messages sent from a particular application are reliably delivered, in the correct sequence, to all applications in a multicast group.

When messages larger than 1Kb are transmitted, OrbixTalk splits the messages into fragments before being sent. OrbixTalk allocates a sequence number to each fragment. The message fragments are stored in memory by the application sending the message then multicast to all applications waiting to receive this message. The applications receiving the message collect all the message fragments and reassemble them. When the last fragment of the message is received and all the fragments for that message have been accepted, the application checks that this is the next expected message. If it is the next expected message, the application passes the message onto the relevant object.

The Reliable Multicast Protocol is a "negative acknowledge" protocol, that is, messages are not explicitly acknowledged. The application sending messages periodically sends out an information message to indicate the sequence number for the last message fragment it sent. If a gap is detected in the incoming message sequence numbers or the application receiving messages misses the last message fragment, it can request the message fragment a number of times before notifying the user software that it has lost some data. The number of times the application can request messages is set using the configuration variables. For more information, refer to Appendix A, "Configuration Parameters".

An application sending messages stores messages, or message fragments, in a memory buffer for a short period of time so that it can resend messages if required. Old messages are deleted from the memory buffer as new messages

are stored. The period of time that messages are stored is set using the configuration variables. For more information, refer to Appendix A, "Configuration Parameters". If an application receiving messages fails, it can re-request the last message sent when it restarts. However, this message may have been deleted from the memory buffer and, therefore, is not available when the application restarts. If this is a problem for a particular application, OrbixTalk MessageStore can be used to store messages persistently allowing an application receiving messages to request playback of any messages that have been missed. For more information, refer to Chapter 3 "OrbixTalk MessageStore" on page 13.

IP Multicast Addresses Details

An IP multicast address (Class D Internet address) consists of a 32-bit number; the high order 4 bits are `1110` which identify the Internet address as an IP multicast address; the remaining 28 bits contain the multicast group ID. Thus, IP multicast addresses are in the range `224.0.0.0` to `239.255.255.255`. The range `224.0.0.0` to `224.0.0.255` is normally reserved and is not used within OrbixTalk.

IP multicast addresses map to ethernet addresses in the range `01:00:5e:00:00:00` to `01:00:5e:7f:ff:ff`. In converting an IP multicast address to an ethernet address, only the first low-order 23 bits of the IP multicast address are used. There is an overlap of 32 IP multicast addresses to each ethernet address. Since the first 23 bits represent approximately 8 million addresses, OrbixTalk only uses addresses which do not overlap.

In practice, there can be system-level limits to how many of these addresses can be used by a specific application. These limits include the number of groups that a process is allowed to join per socket and the number of sockets a process can have open.

3

OrbixTalk MessageStore

This chapter introduces the OrbixTalk MessageStore and the Store and Forward Protocol (otsfp) that add persistent storage of messages and on-demand playback of these messages to the OrbixTalk multicast messaging system.

The OrbixTalk MessageStore can be accessed from applications that use the OrbixTalk API directly or applications that use the CORBA Event Service.

Overview

The OrbixTalk MessageStore provides guaranteed delivery of messages using a Store and Forward Protocol (`otsfp`). Applications send messages to a process, the OrbixTalk MessageStore daemon (`otmsd`), specifying the `otsfp` protocol. The OrbixTalk MessageStore daemon stores the messages in a database and acknowledges receipt of the messages. The messages are then sent to the applications that have registered interest in these messages.

To enable the OrbixTalk MessageStore daemon to store messages, all applications must have a unique application name. There are two types of application name:

- Persistent application name
- Temporary application name

Persistent Application Name

All applications must have a unique application name which enables the OrbixTalk MessageStore daemon to store messages.

For a Supplier Application

Some supplier applications require that message sequence numbers are maintained between invocations. The OrbixTalk MessageStore daemon stores the message sequence numbers for messages which have been successfully sent from these applications and determines the next sequence number for a new message. Applications that require message sequence numbers to be stored are identified by a unique system-wide *persistent application name* which has the following format:

```
//part1/part2/part3/...
```

For example:

```
//Application/one
```

The persistent application name is set by calling `setPersistentAppName`.

For a Consumer Application

A consumer application requires its own state log to store message sequence numbers which have been successfully received from the OrbixTalk MessageStore daemon. If an application fails, it reads the state log at restart and determines the last message it received. This information is sent to the OrbixTalk MessageStore which replays any messages with later sequence numbers. All consumer applications must have a *persistent application name*.

Temporary Supplier Application Name

Some supplier applications using the Store and Forward Protocol (`otsfp`) do not need to maintain state between invocations; for example, an application sending updated prices may not need to keep a log of previous prices. These applications, therefore, do not require message sequence numbers to be stored between invocations. If a supplier application does not set the application name using `setPersistentAppName`, it uses a temporary supplier application name given by the OrbixTalk Directory Enquiries daemon.

Using the OrbixTalk MessageStore

The process of sending, storing and receiving messages through the OrbixTalk MessageStore is as follows:

1. An application that uses a persistent application name sends a request to the OrbixTalk MessageStore to obtain the current state of its message store; that is, it requests the next message sequence number.
2. The application then sends messages to the OrbixTalk MessageStore daemon (`otmsd`) specifying the `otsfp` protocol.
3. The OrbixTalk MessageStore daemon stores the messages in a MessageStore database (duplicate messages are ignored).

   Messages are identified and stored using the Topic name, Application name and message sequence number. The OrbixTalk MessageStore daemon maintains the sequence numbers of messages received from all applications—there is no need for suppliers to maintain a state log.
4. When a message is stored, the OrbixTalk MessageStore daemon sends an acknowledgment to the application sending the message.

   The application tries to re-send messages if it does not receive an acknowledgment within a configurable time period. The number of times the application tries to re-send messages and the time period are set using configuration variables. For more information, refer to Appendix A, “Configuration Parameters”.
5. The OrbixTalk MessageStore daemon uses the `otrmp` protocol to forward messages to the group of applications that have registered interest in these messages. The OrbixTalk MessageStore daemon does not expect an acknowledgment from the applications receiving the messages.
6. The OrbixTalk MessageStore daemon periodically sends a status message containing information about the last message sent on each topic.

   Each application receiving messages detects missing messages by finding gaps in the sequence numbers of messages received or by detecting a difference between the sequence numbers in its state log and the sequence numbers in the status message. If the application detects a missing message, it can request the OrbixTalk MessageStore to play back the message. The time interval for sending status messages is set using the

configuration variables. For more information, refer to Appendix A, "Configuration Parameters".

The OrbixTalk MessageStore also serves as an audit log and can be useful during debugging and testing.

Figure 3.1 illustrates the overall architecture of applications using OrbixTalk MessageStore.

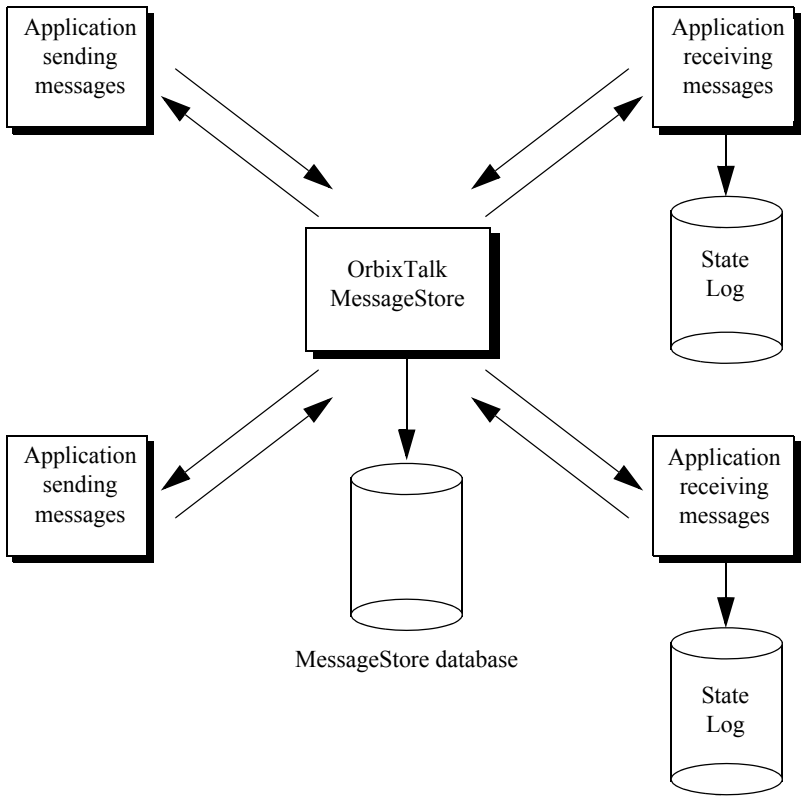

Figure 3.1: *Overall Architecture of the OrbixTalk MessageStore*

Part II

Developing OrbixTalk Applications

4

How OrbixTalk Works

This chapter describes how applications communicate using OrbixTalk.

Overview

OrbixTalk enables applications to communicate using messages. In OrbixTalk, applications sending messages and applications receiving messages are decoupled and communication is asynchronous.

In OrbixTalk, applications sending messages are called *suppliers* and applications receiving messages are called *consumers*. (In previous versions of OrbixTalk these applications were called *talkers* and *listeners* respectively.) Any particular stream of messages is uni-directional, from one or more suppliers to one or more consumers. Thus, a single message stream can simultaneously have more than one supplier and more than one consumer. In general, M suppliers can issue messages via the same message stream to N consumers, without any of the suppliers and consumers having explicit knowledge of each other. One of the advantages of this approach is that new suppliers and consumers can be added easily. The supplier does not have to maintain a list of consumers.

In OrbixTalk, events are delivered to objects as messages. Events of particular types are identified by an OrbixTalk *Topic Name* and a consumer specifies events of interest by informing OrbixTalk about the relevant Topic Names.

OrbixTalk implements this approach using the CORBA Event Service. Details of the Event Service are provided in Chapter 5"The CORBA Event Service".

OrbixTalk Topic Names

OrbixTalk enables shared information to be organized into a hierarchical structure of topics. Each topic is identified by an OrbixTalk *Topic Name*. In this way, an application can determine the information it is interested in and inform OrbixTalk using the Topic Name.

An example of a Topic Name is:

```
"otrmp//iona/teleconf/10.00am"
```

The format of a Topic Name is similar to the Uniform Resource Locator (URL) used by the World Wide Web. The first part of the name identifies the communications protocol. In the example, `otrmp` is the OrbixTalk Reliable Multicast Protocol, part of OrbixTalk's transport service. The OrbixTalk Reliable Multicast Protocol is OrbixTalk's default protocol. To use the OrbixTalk MessageStore, the OrbixTalk Store and Forward Protocol (`otsfp`) must be specified.

The rest of the name is hierarchically organized, allowing a great deal of flexibility in organizing the OrbixTalk name space. In the example, the Topic Name identifies a teleconference organized by IONA and held at 10.00 a.m.

Using Wildcards with Topic Names

A consumer can listen on a wildcarded topic to register interest in all messages on a set of (possibly unknown) topics.

Consider the following list of topics:

```
otrmp//Stock/Iona/US
otrmp//Stock/Iona/Europe
otrmp//Stock/Sun/US
otrmp//Stock/Sun/Europe
otrmp//Stock/IBM/US
otrmp//Stock/IBM/Europe/East
otrmp//Stock/IBM/Europe/West
```

There are two ways that wildcards can be used to match topics:

1. Using an asterisk (*) as one or more of the name parts of a topic. The asterisk matches any name part in the *same* position. For example:

   ```
   otrmp//Stock/*/US
   ```

   This matches the following:

```
otrmp//Stock/Iona/US
otrmp//Stock/Sun/US
otrmp//Stock/IBM/US
```

2. Using a double asterisk (**) to match the remainder of a topic name. For example:

   ```
   otrmp//Stock/IBM/**
   ```

   This matches the following:

   ```
   otrmp//Stock/IBM/US
   otrmp//Stock/IBM/Europe/East
   otrmp//Stock/IBM/Europe/West
   ```

Communication between Suppliers and Consumers

Suppliers and consumers communicate by sending and receiving messages on a specified Topic Name. The Topic Name is translated into an IP multicast address by the OrbixTalk Directory Enquiries daemon.

OrbixTalk Directory Enquiries Daemon

OrbixTalk uses meaningful hierarchical Topic Names to identify the information groups. The OrbixTalk Directory Enquiries daemon translates the Topic Names into IP multicast addresses. The OrbixTalk Directory Enquiries daemon is only contacted the first time a Topic Name is used by an application so it does not become a performance bottleneck. For more information about IP multicast addresses, see Chapter 2 "The OrbixTalk Reliable Multicast Protocol" on page 9.

There are two OrbixTalk Directory Enquiries daemons available:

1. The basic OrbixTalk Directory Enquiries daemon (`otd`).
2. The Directory Enquiries daemon, `otdsm` enables you to view information about Topic and Application Names stored in the OrbixTalk Directory Enquiries daemon.

Sending Messages

To send messages on a Topic Name, the supplier application creates a proxy object and registers the proxy object as a supplier with the OrbixTalk Directory Enquiries daemon. Invocations are made on the proxy object and all communications via a proxy object use the OrbixTalk multicast transport service. In this way, suppliers are not linked to the consumer objects.

When a supplier application sends a message on a specific Topic Name, the supplier checks to see if the Topic Name has been translated into an IP multicast address. If not, a request is sent to the OrbixTalk Directory Enquiries daemon requesting the IP multicast address for the Topic Name. If the mapping between the Topic Name and IP multicast address exists, the OrbixTalk Directory Enquiries daemon returns the IP multicast address. If the mapping between the Topic Name and IP multicast address does not exist, the OrbixTalk Directory Enquiries daemon allocates a new IP multicast address. The supplier application then sends the message using the IP multicast address.

Receiving Messages

A consumer application requests information by specifying the Topic Name to OrbixTalk. The OrbixTalk Directory Enquiries daemon translates the Topic Name into an IP multicast address. Messages arriving on this IP multicast address are passed to consumers specifying the Topic Name. If there are multiple consumers listening on a Topic Name, all of them receive the same information.

Figure 4.1 summarizes the communication between a supplier and a consumer via the OrbixTalk Directory Enquiries daemon and the OrbixTalk multicast transport service.

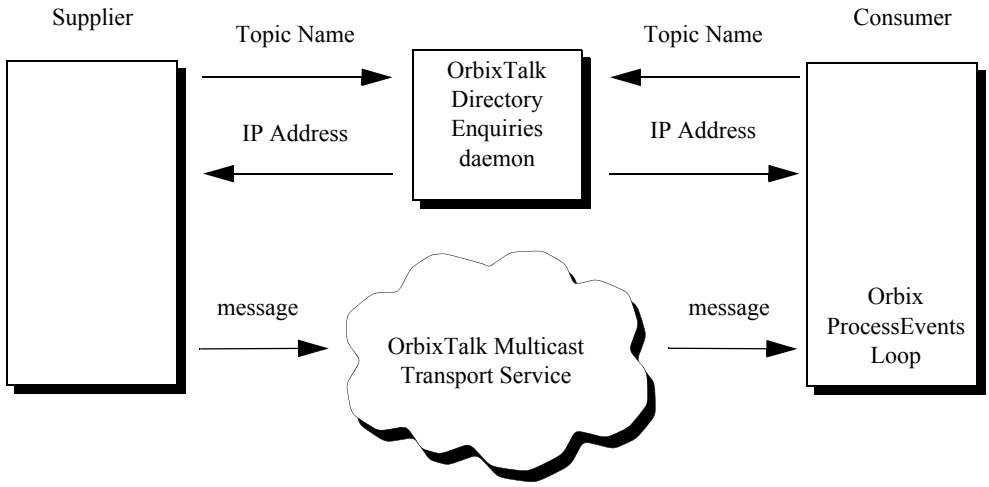

Figure 4.1: *OrbixTalk Architecture*

OrbixTalk Transport Protocol Stack

The OrbixTalk protocol stack provides three protocol levels as shown in Figure 4.2.

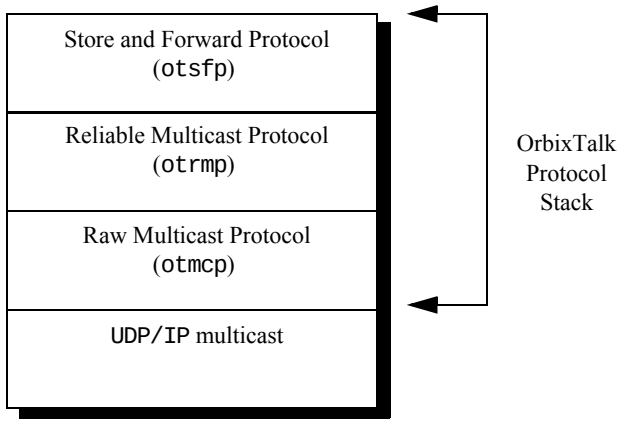

Figure 4.2: *OrbixTalk Protocol Stack*

This section provides information about the OrbixTalk Raw Multicast Protocol (otmcp).

For more information about the Store and Forward Protocol, refer to Chapter 3 "OrbixTalk MessageStore" on page 13. For more information about the Reliable Multicast Protocol, refer to Chapter 2 "The OrbixTalk Reliable Multicast Protocol" on page 9.

OrbixTalk Raw Multicast Protocol (otmcp)

The OrbixTalk Raw Multicast Protocol is a thin layer on top of UDP/IP Multicast providing a light weight form of unreliable multicast. This layer supports the OrbixTalk message format but does not provide fragmentation, reassembly, sequencing, reliable transfer or ordering of packets. Applications using this layer must ensure that the method invocation fits into one OrbixTalk packet which is 1280 bytes long, using a small number of parameters consisting of basic types or simple structs.

For example, you could use this layer to provide a "heart beating" mechanism to implement a fault tolerant system:

- Create a print server application that sends a heartbeat every 5 seconds on the `otmcp` topic `otmcp//HeartBeat/PrintServer`.
- Create a monitoring application that receives messages on the `otmcp` topic `otmcp//HeartBeat/PrintServer`.

If the monitoring application misses 5 heartbeats in a row it assumes that the print server application has died and launches another print server application to take over.

OrbixTalk Transport Implementation

The implementation of the OrbixTalk Transport is fully multi-threaded to achieve maximum performance for the OrbixTalk Transport protocol stack. An OrbixTalk Topic Name is mapped to an IP multicast address. For each IP multicast address used there is a corresponding socket set which handles incoming and outgoing message traffic on that IP multicast address. This socket set is handled by its own thread set to ensure that the socket traffic is efficiently serviced. This thread set includes a timer event loop thread to handle the timers specific to the socket set. OrbixTalk also provides a user timer events loop thread to implement asynchronous applications via OrbixTalk timer events. Using a Multi-Threaded OrbixTalk Transport protocol stack removes the need for you to *drive* the event loop thus ensuring a much more efficient protocol. For more information about the timer event loop and user timer events loop, refer to “OrbixTalk Timer Events” on page 230.

5

The CORBA Event Service

OrbixTalk is modeled on the CORBA Event Service. This specification defines a model of communication that allows an application to send an event that will be received by any number of objects. The model provides two approaches to initiating event communication. For each of these approaches, event communication can take two forms. This chapter introduces the terminology and concepts that are used throughout this guide.

OrbixTalk implements the CORBA Event Service specification to provide multicast messaging. This specification defines a model for communications between ORB applications that supplements the direct operation call system that client/server applications normally use.

This chapter introduces the basic concepts of the CORBA Event Service communications model. Later chapters will describe the programming interface in detail and show how to implement applications that use the CORBA Event Service for multicast messaging with OrbixTalk.

Communications using the CORBA Event Service

Figure 5.1 illustrates the standard CORBA model for communication between distributed applications.

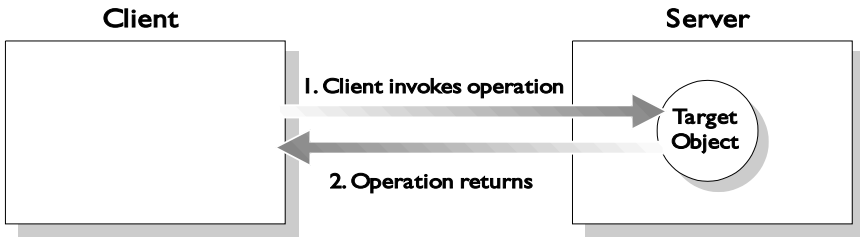

Figure 5.1: *CORBA Model for Basic Client/Server Communications*

In this model, a client application calls an IDL operation on a specified object in a server. The client waits for the call to complete and then receives confirmation of the return status. For any operation call there is a single client and a single server, and each must be available for the call to succeed.

This simple, one-to-one communication model is fundamental to the CORBA architecture. However, some ORB applications need a more complex, indirect communication style. The CORBA Event Service defines a communication model that allows an application to send a message to objects in other applications without any knowledge about the objects that receive the message.

The CORBA Event Service introduces the concept of *events* to CORBA communications. An event originates at an event *supplier* and is transferred to any number of event *consumers*. Suppliers and consumers are completely decoupled: a supplier has no knowledge of the number of consumers or their identities, and consumers have no knowledge of which supplier generated a given event.

In order to support this model, the CORBA Event Service introduces to CORBA a new architectural element, called an *event channel*. An event channel mediates the transfer of events between the suppliers and consumers as follows:

1. The event channel allows consumers to register interest in events, and stores this registration information.
2. The channel accepts incoming events from suppliers.
3. The channel forwards supplier-generated events to registered consumers.

Suppliers and consumers connect to the event channel and not directly to each other (Figure 5.2). From a supplier's perspective, the event channel appears as a single consumer; from a consumer's perspective, the event channel appears as a single supplier. In this way, the event channel decouples suppliers and consumers.

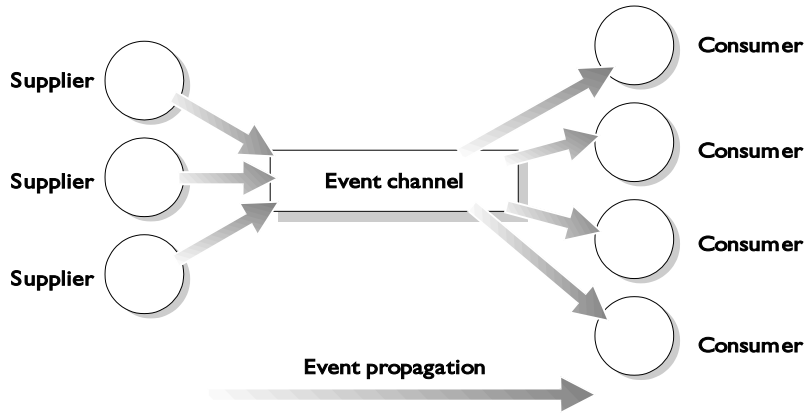

Figure 5.2: *Suppliers and Consumers Communicating through an Event Channel*

Any number of suppliers can issue events to any number of consumers using a single event channel. There is no correlation between the number of suppliers and the number of consumers, and new suppliers and consumers can be easily added to the system. In addition, any supplier or consumer can connect to more than one event channel.

A typical example that uses an event-based communication model is that of a spreadsheet cell. Many documents may be linked to a spreadsheet cell and these documents need to be notified when the cell value changes. However, the spreadsheet software should not need knowledge of each document linked to the cell. When the cell value changes, the spreadsheet software should be able to issue an event which is automatically forwarded to each connected document.

CORBA defines the Event Service at a level above the ORB architecture. Suppliers, consumers and event channels may be implemented as ORB applications, while events are defined using standard IDL operation calls. Suppliers, consumers and event channels each implement clearly defined IDL interfaces that support the steps required to transfer events in a distributed system.

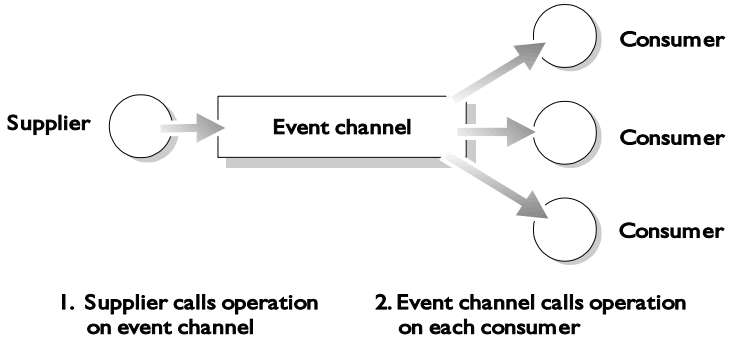

Figure 5.3: *An Example Implementation of Event Propagation*

Figure 5.3 illustrates an example implementation of event propagation in a CORBA system. In this example, suppliers are implemented as CORBA clients; the event channel and consumers are implemented as CORBA servers. An event occurs when a supplier invokes a clearly defined IDL operation on an object in the event channel application. The event channel propagates the event by invoking a similar operation on objects in each of the consumer servers. To make this possible, the event channel application stores a reference to each of the consumer objects, for example, in an internal list.

This is not the only way in which the concept of events can map to a CORBA system. In particular, the CORBA Event Service identifies two approaches to initiating the propagation of events, and these affect the implementation architecture. “Initiating Event Communication” on page 31 addresses this topic in detail.

“Types of Event Communication” on page 35 discusses how events can map to IDL operation calls, and describes how you can associate data with an event using IDL operation parameters.

Initiating Event Communication

CORBA specifies two approaches to initiating the transfer of events between suppliers and consumers. These approaches are called the *Push model* and the *Pull model*. In the Push model, suppliers initiate the transfer of events by sending those events to consumers. In the Pull model, consumers initiate the transfer of events by requesting those events from suppliers.

This section illustrates each approach in turn, and then describes how these models can be mixed in a single system.

The Push Model

In the Push model, a supplier generates events and actively passes them to a consumer. In this model, a consumer passively waits for events to arrive. Conceptually, suppliers in the Push model correspond to clients in normal CORBA applications, and consumers correspond to servers.

Figure 5.4 illustrates a Push model architecture in which push suppliers communicate with push consumers through an event channel.

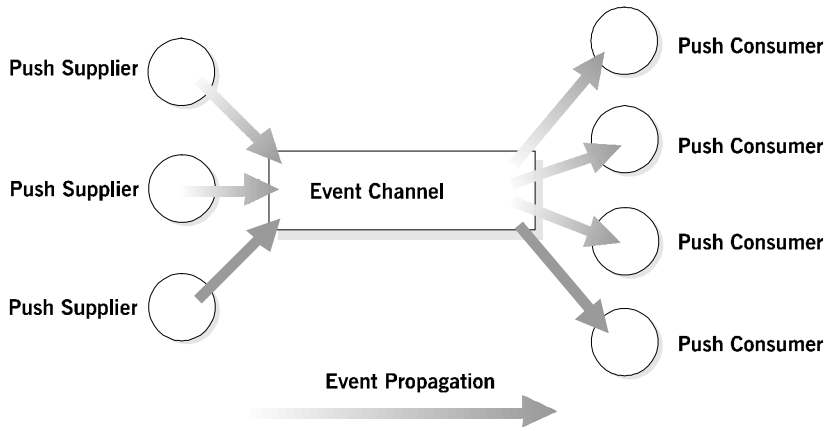

Figure 5.4: *Push Model Suppliers and Consumers Communicating through an Event Channel*

In this architecture, a supplier initiates the transfer of an event by invoking an IDL operation on an object in the event channel. The event channel invokes a similar operation on an object in each consumer that has registered with the channel.

The Pull Model

In the Pull model, a consumer actively requests that a supplier generate an event. In this model, the supplier waits for a pull request to arrive. When a pull request arrives, event data is generated by the supplier and returned to the pulling consumer. Conceptually, consumers in the Pull model correspond to clients in normal CORBA applications and suppliers correspond to servers.

Figure 5.5 illustrates a Pull model architecture in which pull consumers communicate with pull suppliers through an event channel.

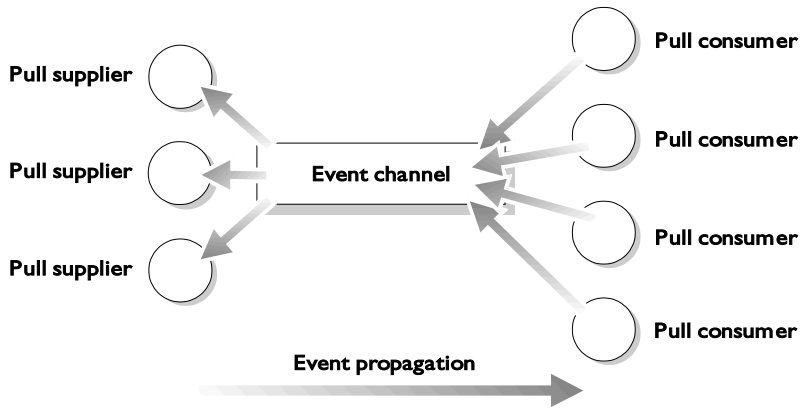

Figure 5.5: *Pull Model Suppliers and Consumers Communicating through an Event Channel*

In this architecture, a consumer initiates the transfer of an event by invoking an IDL operation on an object in the event channel application. The event channel then invokes a similar operation on an object in each supplier. The event data is returned from the supplier to the event channel and then from the channel to the consumer which initiated the transfer.

Mixing the Push and Pull Models in a Single System

Because suppliers and consumers are completely decoupled by an event channel, the Push and Pull models can be mixed in a single system. For example, suppliers may connect to an event channel using the Push model, while consumers connect using the Pull model as shown in Figure 5.6.

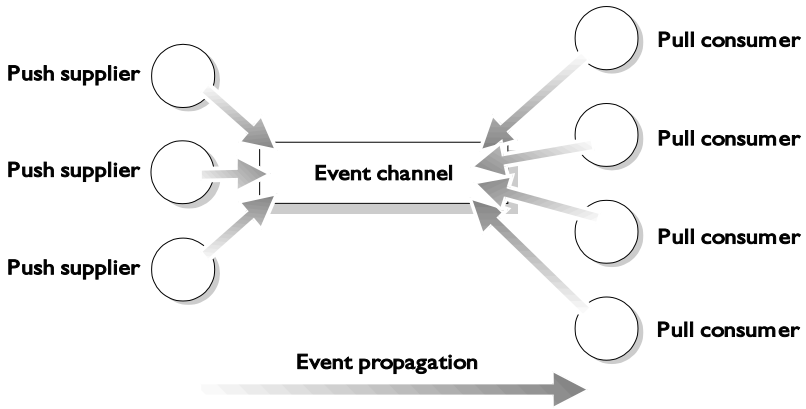

Figure 5.6: *Push Model Suppliers and Pull Model Consumers in a Single System*

In this case, both suppliers and consumers must participate in initiating event transfer. A supplier invokes an operation on an object in the event channel to transfer an event to the channel. A consumer then invokes another operation on an event channel object to transfer the event data from the channel. Unlike the case in which consumers connect using the Push model, the event channel takes no initiative in forwarding the event. The event channel stores events supplied by the push suppliers until some pull consumer requests an event, or until a push consumer connects to the event channel.

Types of Event Communication

The CORBA Event Service maps an event to a successfully completed sequence of operation calls. The operations and the sequence of calls are clearly defined for both Push and Pull models, and data about an event can be passed as operation parameters or return values. This data is specific to each application and is generally not interpreted by implementations of the CORBA Event Service, such as OrbixTalk.

Event communication can take one of the two forms, *typed* or *untyped*.

Untyped Event Communication

In untyped event communication, an event is propagated by a series of generic `push()` or `pull()` operation calls. The `push()` operation takes a single parameter which stores the event data. The event data parameter is of type `any`, which allows any IDL defined data type to be passed between suppliers and consumers. The `pull()` operation has no parameters but transmits event data in its return value, which is also of type `any`. Clearly, in both cases, the supplier and consumer applications must agree about the contents of the `any` parameter and return value if this data is to be useful.

Typed Event Communication

In typed event communication, a programmer defines application-specific IDL interfaces through which events are propagated. Rather than using `push()` and `pull()` operations and transmitting data using an `any`, a programmer defines an interface that suppliers and consumers use for the purpose of event communication. The operations defined on the interface may contain parameters defined in any suitable IDL data type. In the Push model, event communication is initiated simply by invoking operations defined on this interface. The Pull model is more complex because event communication is initiated by invoking operations on an interface that is specially constructed from the application-specific interface that the programmer defines. Event communication is initiated by invoking operations on the constructed interface.

The form that event communication takes is independent of the method of initiating event transfer. As a consequence, the Push model can be used to transmit typed events or untyped events, and the Pull model can be used to transmit typed or untyped events.

6

The Programming Interface to the Event Service

The CORBA Event Service specification defines a set of interfaces that support the Push and Pull models of initiating the transfer of events in both typed and untyped format. This chapter gives details of these interfaces. The CORBA Event Service specification defines the roles of consumer, supplier and event channel by describing IDL interfaces that each must support. The operations on these interfaces allow consumers and suppliers to register with an event channel to enable the propagation of events.

The CORBA Event Service includes IDL interfaces for both untyped and typed events in both the Push and Pull event models. This chapter describes in detail the IDL interfaces defined for the CORBA Event Service to support these models.

You can find a complete listing of all interfaces relating to the CORBA Event Service in Appendix C, "CORBA Event Service: IDL Interfaces".

The Programming Interface for Untyped Events

The CORBA Event Service for untyped events defines interfaces for suppliers, consumers and event channels. It also defines a number of administration interfaces that allow suppliers and consumers to register with an event channel to allow the transfer of events between them.

Registration of Suppliers and Consumers with an Event Channel

A supplier connects to an event channel to indicate that it wishes to transfer events to consumers through that channel. A consumer connects to an event channel to register its interest in any events supplied through that channel. When a supplier or consumer no longer wishes to send or receive events, the application may disconnect itself from the event channel. In some cases, the event channel may need to disconnect a supplier or consumer explicitly.

The CORBA Event Service defines a set of interfaces that supports untyped event transfer using the Push and Pull models. These interfaces are described in the remainder of this section.

The Push Model for Untyped Events

Four IDL interfaces support connection to and disconnection from event channels using the Push model:

```
PushSupplier
PushConsumer
ProxyPushConsumer
ProxyPushSupplier
```

The interfaces `PushSupplier` and `ProxyPushConsumer` allow suppliers to supply events to an event channel.

The interfaces `PushConsumer` and `ProxyPushSupplier` are specific to consumers, allowing them to receive events from an event channel.

These four interfaces are defined in IDL as follows:

```
// IDL
module CosEventComm {
   exception Disconnected {
   };

   interface PushConsumer {
      void push (in any data) raises (Disconnected);
      void disconnect_push_consumer ();
   };

   interface PushSupplier {
      void disconnect_push_supplier();
   };
};


module CosEventChannelAdmin {
   exception AlreadyConnected {
   };

   exception TypeError {
   };

   interface ProxyPushConsumer : CosEventComm::PushConsumer {
      void connect_push_supplier (
         in CosEventComm::PushSupplier push_supplier)
         raises (AlreadyConnected);
   };

   interface ProxyPushSupplier : CosEventComm::PushSupplier {
      void connect_push_consumer (
         in CosEventComm::PushConsumer push_consumer)
         raises (AlreadyConnected, TypeError);
   };

   ...
};
```

Connecting a Supplier

A supplier initiates connection to an event channel by obtaining a reference to an object of type `ProxyPushConsumer` in the channel. The supplier application may wish to be notified if the event channel terminates the connection. If so, the supplier then invokes the operation `connect_push_supplier()` on that object, passing a reference to an object of type `PushSupplier` as an operation parameter. If the `ProxyPushConsumer` is already connected to a `PushSupplier`, `connect_push_supplier()` will raise the exception `AlreadyConnected`.

Connecting a Consumer

A consumer first obtains a reference to a `ProxyPushSupplier` object implemented in the event channel. In order to register its interest in events from the channel, the consumer then invokes the operation `connect_push_consumer()` on the `ProxyPushSupplier` object. The consumer passes a reference to an object of type `PushConsumer` to the operation call.

If `ProxyPushSupplier` is already connected to a `PushConsumer`, `connect_push_consumer()` will raise the exception `AlreadyConnected`.

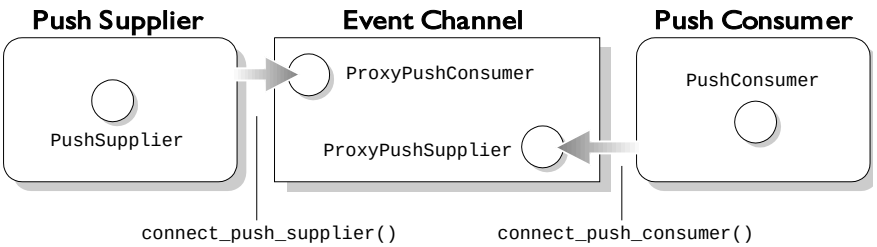

Figure 6.1: *Push Supplier and Push Consumer Connecting to an Event Channel in the Untyped Model*

Figure 6.1 illustrates how a supplier and consumer connect to an event channel. There are no dependencies between the connection of the supplier and the connection of the consumer.

The Pull Model for Untyped Events

A similar set of IDL interfaces supports connection to and disconnection from event channels in the Pull model. These interfaces are:

```
PullSupplier
PullConsumer
ProxyPullConsumer
ProxyPullSupplier
```

The interfaces `PullConsumer` and `ProxyPullSupplier` allow consumers to request events from an event channel.

The interfaces `PullSupplier` and `ProxyPullConsumer` allow an event channel to request events from suppliers.

The Pull model interfaces are defined in IDL as follows:

```
// IDL
module CosEventComm {
   exception Disconnected {
   };

   interface PullSupplier {
      any pull () raises (Disconnected);
      any try_pull (out boolean has_event) raises (Disconnected);
      void disconnect_pull_supplier();
   };

   interface PullConsumer {
      void disconnect_pull_consumer ();
   };
};


module CosEventChannelAdmin {
   exception AlreadyConnected {
   };

   exception TypeError {
```

```
    };

    interface ProxyPullSupplier : CosEventComm::PullSupplier {
       void connect_pull_consumer (
          in CosEventComm::PullConsumer pull_consumer)
          raises (AlreadyConnected);
    };

    interface ProxyPullConsumer : CosEventComm::PullConsumer {
       void connect_pull_supplier (
          in CosEventComm::PushSupplier pull_supplier)
          raises (AlreadyConnected, TypeError);
    };

    ...
};
```

Connecting a Consumer

In the Pull model, the transfer of events is initiated by consumers. A consumer initiates connection to an event channel by obtaining a reference to an object of type `ProxyPullSupplier` in the channel. The consumer application may wish to be notified if the event channel terminates the connection. If so, it invokes the operation `connect_pull_consumer()` on the `ProxyPullSupplier` object, passing a reference to an object of type `PullConsumer` as an operation parameter. If the `ProxyPullSupplier` is already connected to a `PullConsumer`, `connect_pull_consumer()` raises the exception `AlreadyConnected`.

Connecting a Supplier

To connect to an event channel, a pull supplier first obtains a reference to a `ProxyPullConsumer` object implemented in the event channel. The supplier then invokes the operation `connect_pull_supplier()` on the `ProxyPullConsumer` object, passing a reference to an object of type `PullSupplier` as the operation parameter. If the `ProxyPullConsumer` is already connected to a `PullSupplier`, `connect_pull_supplier()` raises the exception `AlreadyConnected`.

Transfer of Untyped Events through an Event Channel

The transfer of events from a supplier through an event channel to a consumer follows a simple pattern. Events originate at a supplier. In the Push model, a supplier pushes events into the event channel which in turn pushes the events to registered consumers. In the Pull model, consumers take the active role by requesting events from the event channel; the event channel, in turn, requests events from registered suppliers. Both methods of transfer are described for *untyped* events in the remainder of this section.

The Push Model

The supplier initiates event transfer by invoking the operation `push()` on a `ProxyPushConsumer` object in the event channel, passing the event data as a parameter of type `any`. The event channel then invokes a `push()` operation on the `PushConsumer` object in each registered consumer, again passing the event data as an operation parameter. Conceptually, this transfer is as shown in Figure 6.2.

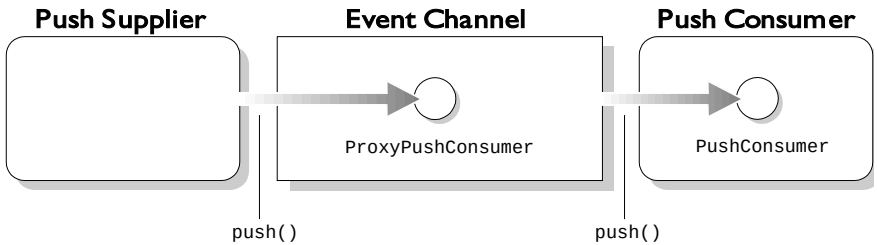

Figure 6.2: *Transfer of an Event through an Event Channel to a Consumer using the Untyped Push Model*

Note that the supplier views the event channel as a single consumer and has no knowledge of the actual consumers. Likewise, the consumer views the event channel as a single supplier. In this way, the channel decouples the supplier and consumer.

The Pull Model

The consumer initiates event transfer in the Pull model. The consumer initiates event transfer in one of two ways as described below.

- `pull()`

  The consumer invokes the `pull()` operation on a `ProxyPullSupplier` object in the event channel.

  The event channel, if it does not already have an event, invokes a `pull()` operation on the `PullSupplier` object in each registered supplier.

  The `pull()` operation blocks until an event is available; the operation then returns the event data in its return value which is of type `any`. Thus, the consumer application blocks until the event channel can supply an event. The event channel, in turn, blocks until some supplier supplies an event to the channel. .

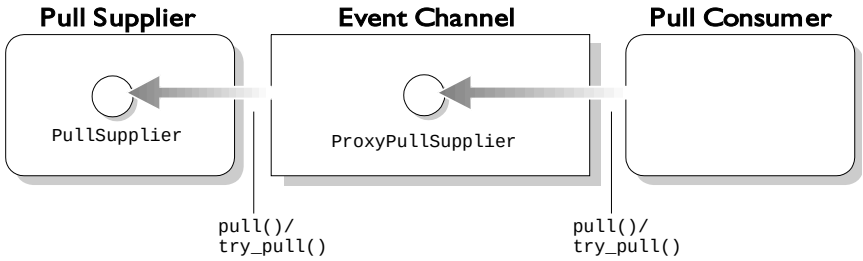

Figure 6.3: *Transfer of an Event through an Event Channel to a Consumer using the Untyped Pull Model*

- `try_pull()`

  The consumer invokes the `try_pull()` operation on a `ProxyPullSupplier` object in the event channel.

  The event channel, in turn, invokes a `try_pull()` operation on the `PullSupplier` object in each registered supplier.

  If no supplier has an event available, `try_pull()` sets its boolean `has_event` parameter to false and returns immediately. If an event is available from some supplier, `try_pull()` sets the `has_event` parameter to true and returns the event data in its return value which is of type `any`.

Conceptually, the transfer of an event using the Pull model is as shown in Figure 6.3

Note that, as in the Push model, the channel decouples suppliers and consumers. The consumer views the event channel as a single supplier and has no knowledge of the actual suppliers. Likewise, the supplier views the event channel as a single consumer.

Event Channel Administration Interfaces

The CORBA Event Service specification defines a set of interfaces that support event channel administration. The role of these interfaces is to allow a supplier or consumer to make initial contact with an event channel and to provide a set of standardized operations so that a supplier may obtain a `ProxyPushConsumer` or `ProxyPullConsumer` and a consumer may obtain a `ProxyPushSupplier` or `ProxyPullSupplier` object reference.

Each event channel supports the interface `EventChannel`, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
   ...

   interface EventChannel {
      ConsumerAdmin for_consumers ();
      SupplierAdmin for_suppliers ();
      void destroy ();
   };
};
```

If a supplier or consumer wishes to connect to an event channel, it must first obtain a reference to an `EventChannel` object in that channel. Typically, the event channel will publish a reference for this object, for example using the CORBA Naming Service.

A supplier then invokes the operation `for_suppliers()` on the `EventChannel` object. This operation returns a reference to an object of type `SupplierAdmin`, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
   interface SupplierAdmin {
      ProxyPushConsumer obtain_push_consumer ();
      ProxyPullConsumer obtain_pull_consumer ();
   };

   ...
};
```

To obtain a reference to a `ProxyPushConsumer` object in the event channel, the supplier invokes the operation `obtain_push_consumer()` on the `SupplierAdmin` object. At this point, the supplier is ready to connect to the channel and begin transferring events using the Push model.

The supplier invokes the operation `obtain_pull_consumer()` on the `SupplierAdmin` object if it wishes to obtain a `ProxyPullConsumer`. The supplier is then ready to connect to the channel and to transfer events using the Pull model.

Similarly, a consumer invokes the operation `for_consumers()` on an `EventChannel` object in order to obtain a reference to an object of type `ConsumerAdmin`, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
   interface ConsumerAdmin {
      ProxyPushSupplier obtain_push_supplier ();
      ProxyPullSupplier obtain_pull_supplier ();
   };

   ...
};
```

If the consumer is using the Push model, it then invokes the operation `obtain_push_supplier()` to obtain a reference to a `ProxyPushSupplier`. If the consumer is using the Pull model, it invokes the operation `obtain_pull_supplier()` to obtain a reference to a `ProxyPullSupplier` object in the event channel.

The consumer is then free to register its interest in events propagated through the channel.

The Programming Interface for Typed Events

As described in "Types of Event Communication" on page 35, events can be communicated in untyped form or in typed form. As OrbixTalk supports the Push model for Typed events, this section describes the Push model only.

Using typed event communication, you can define application-specific IDL interfaces through which events can be propagated. You are not restricted to using the operation `push()` to transfer events, and you do not have to pack operation parameters into an IDL `any`.

The operations you specify in your interfaces may define `in` parameters to allow suppliers to transmit event data. However, since event propagation is uni-directional, these operations may not define `inout` or `out` parameters; they must have a `void` return value and may not have a `raises` clause. These restrictions are the same as the restrictions on `oneway` operations. However, you do not have to define the operations to be `oneway`.

The model for typed event communication closely follows the model for untyped events. Typed suppliers connect to a proxy consumer in the event channel and typed consumers connect to a proxy supplier.

Suppliers and consumers must agree on the interface they will use to transfer events. To illustrate this, recall the example of the spreadsheet in "Communications using the CORBA Event Service" on page 27. Many documents can be linked to a spreadsheet cell and these need to be notified of changes to the cell value. The spreadsheet software notifies interested documents of a change to a cell value by generating an event that is forwarded to each connected document. An interface that supports notification of changes to a cell value might be defined as follows:

```
// IDL
interface SpreadsheetCell {
   void value_changed (in float new_value);
   ...
};
```

In this example, documents that are linked to a cell are notified by the spreadsheet software which supplies the event `SpreadsheetCell::value_changed()` whenever the value of a cell changes. The interface `SpreadsheetCell` may define other operations that may be used to supply events to connected documents.

Registration of Suppliers and Consumers with a Typed Event Channel

This section describes how suppliers and consumers register with an event channel in the typed model. The sequence of steps is very similar to that described for the untyped model.

The Typed Push Model

Four IDL interfaces support connection to and disconnection from event channels using the typed Push model:

```
PushSupplier
TypedPushConsumer
ProxyPushSupplier
TypedProxyPushConsumer
```

The interfaces `PushSupplier` and `TypedProxyPushConsumer` allow suppliers to supply events to an event channel.

The interfaces `TypedPushConsumer` and `ProxyPushSupplier` allow consumers to receive events from an event channel.

`PushSupplier` and `ProxyPushSupplier` are as described for the untyped Push model in "Registration of Suppliers and Consumers with an Event Channel" on page 38.

The interfaces `TypedPushConsumer` and `TypedProxyPushConsumer` inherit from their counterparts in the untyped Push model. They are defined as follows:

```
// IDL
module CosTypedEventComm {
   interface TypedPushConsumer : CosEventComm::PushConsumer {
      Object get_typed_consumer ();
   };
};
```

```
module CosTypedEventChannelAdmin {
   interface TypedProxyPushConsumer :
         CosEventChannelAdmin::ProxyPushConsumer,
         CosTypedEventComm::TypedPushConsumer {
   };
};
```

A typed push supplier initiates connection to an event channel by obtaining a reference to a `TypedProxyPushConsumer` object in the event channel. The supplier invokes the operation `connect_push_supplier()` on the `TypedProxyPushConsumer` object, passing a reference to an object of type `PushSupplier` as an operation parameter.

A typed push consumer obtains a reference to a `ProxyPushSupplier` object in the event channel and invokes the operation `connect_push_consumer()` on that object, passing an object of type `TypedPushConsumer` as the operation parameter. Figure 6.4 illustrates how a supplier and consumer connect to the event channel.

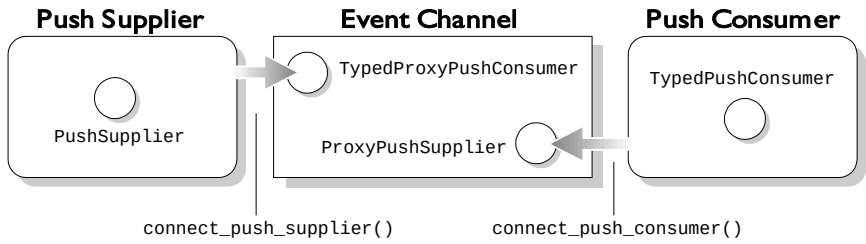

Figure 6.4: *Push Supplier and Push Consumer Connecting to an Event Channel using the Typed Model*

Transfer of Typed Events Through an Event Channel

Once connected to an event channel, suppliers initiate the transfer of typed events in the Push model.

The Typed Push Model

At this point, the typed push supplier is connected to the event channel as described in "Registration of Suppliers and Consumers with a Typed Event Channel" on page 48; specifically, it is connected to a `TypedProxyPushConsumer` object in the event channel.

The `TypedProxyPushConsumer` object is specific to the type of events supplied by the supplier; that is, the supplier and the event channel agree on the type of events supplied by the supplier and accepted by the channel. This agreement is reached when the `TypedProxyPushConsumer` object is set up using the event channel administration interfaces; these interfaces are described in "Typed Event Channel Administration Interfaces" on page 51.

To set up the transfer of events into the channel, the supplier invokes the operation `get_typed_consumer()` on the `TypedProxyPushConsumer` object. The operation `get_typed_consumer()` returns an object reference that supports the interface for which the `TypedProxyPushConsumer` was created. In this example, this is the interface `SpreadsheetCell`. The return type from `get_typed_consumer()` is `CORBA::Object`. Therefore, the supplier must narrow this object reference to obtain a reference of the type for which it supplies events—in this case, `SpreadsheetCell`.

Having obtained this object reference, the supplier supplies events to the event channel simply by invoking operations defined in interface `SpreadsheetCell` on the object reference returned by `get_typed_consumer()`. Data associated with the event, if any, is supplied using the operations' `in` parameters. Conceptually, the transfer is as shown in Figure 6.5, where `SpreadsheetCell::value_changed()` events are generated by the supplier.

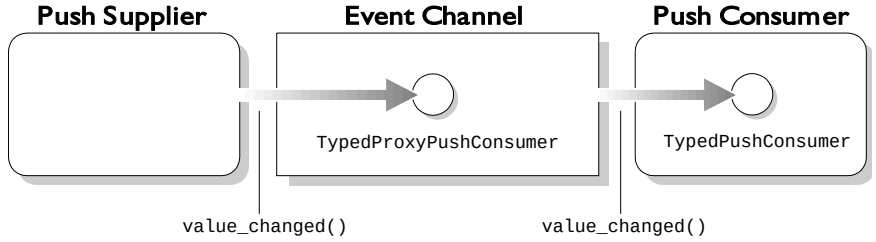

Figure 6.5: *Transfer of an Event Through an Event Channel to a Consumer Using the Typed Push Model*

Typed push suppliers send messages to consumers even though these suppliers do not know anything about the consumers that receive these messages. The flow of information is unidirectional—from suppliers to consumers. Therefore, data associated with an event can be sent by a supplier in an operation's `in` parameters, but no data can be returned because no operation reply can be received by the supplier. Thus, operations invoked by the supplier may not have `inout` or `out` parameters; they must have a `void` return value; and they cannot have a `raises` clause. (These same restrictions apply to `oneway` operations in the standard CORBA model.)

Typed Event Channel Administration Interfaces

To support typed event communication, the CORBA Event Service specification provides a set of administration interfaces similar to those provided for the administration of untyped event channels. Where appropriate, these interfaces use IDL inheritance to indicate that they are specializations of corresponding interfaces in the untyped model.

The interface to a typed event channel is described by the interface `TypedEventChannel`, which is defined as follows:

```
// IDL
module CosTypedEventChannelAdmin {

   interface TypedEventChannel {
      TypedConsumerAdmin for_consumers ();
      TypedSupplierAdmin for_suppliers ();
      void destroy ();
   };

};
```

To connect to a typed event channel, a supplier or consumer must first obtain a reference to a `TypedEventChannel` object in that channel. As for the untyped model, the event channel will typically publish a reference for this object, for example, using the CORBA Naming Service.

A supplier then invokes the operation `for_suppliers()` on the `TypedEventChannel` object. This operation returns a reference to an object of type `TypedSupplierAdmin`, which is defined as follows:

```
// IDL
```

```
module CosTypedEventChannelAdmin {

   exception InterfaceNotSupported {
   };

   exception NoSuchImplementation {
   };

   typedef string Key;

   interface TypedSupplierAdmin :
      CosEventChannelAdmin::SupplierAdmin {

      TypedProxyPushConsumer obtain_typed_push_consumer (
               in Key supported_interface)
               raises (InterfaceNotSupported);

      ProxyPullConsumer obtain_typed_pull_consumer (
               in Key uses_interface)
               raises (NoSuchImplementation);
   };

   ...
};
```

The next step occurs when a push supplier invokes the operation `obtain_typed_push_consumer()` on the `TypedSupplierAdmin` object to obtain a reference to a `TypedProxyPushConsumer` object in the event channel.

Once the supplier has a `TypedSupplierAdmin` object, it is ready to connect to the channel and begin transferring events.

Similarly, a consumer invokes the operation `for_consumers()` on an `TypedEventChannel` object to obtain a reference to an object of type `TypedConsumerAdmin`, which is defined as follows:

```
// IDL
module CosTypedEventChannelAdmin {
   exception InterfaceNotSupported {
   };

   exception NoSuchImplementation {
   };
```

```
typedef string Key;

interface TypedConsumerAdmin :
        CosEventChannelAdmin::ConsumerAdmin {

   ProxyPushSupplier
   obtain_typed_push_supplier (
            in Key uses_interface)
            raises (NoSuchImplementation);
};
      ...
   };
```

The push consumer invokes the operation `obtain_typed_push_supplier()` to obtain a reference to a `ProxyPushSupplier`. The consumer is then free to register its interest in events propagated through the channel.

7

Programming with the Untyped Push Model

To illustrate the Push model communicating untyped events, this chapter develops a simple application.

As described in Chapter 5, "The CORBA Event Service", OrbixTalk allows you to develop Object Request Broker (ORB) applications that communicate using the CORBA Event Service communications model. From a programmer's perspective, the event channel is the key element of a CORBA Event Service application.

The OrbixTalk IIOP Gateway provides event channels for multicast suppliers and consumers. Any IIOP-compliant application, such as OrbixWeb, can therefore make use of multicast functionality.

You can also build C++ suppliers and consumers that use the multicast protocols directly by incorporating the colocated C++ library functions included with OrbixTalk. This subject is discussed in Chapter 10, "The OrbixTalk Events Library".

This chapter describes an example ORB application that illustrates how you can use OrbixTalk to develop Push model suppliers and consumers that communicate untyped events through event channels implemented by the IIOP Gateway.

Overview of an Example Application

The example described in this chapter consists of a push supplier and a push consumer, each of which connects to a single event channel. The supplier repeatedly pushes an event to the event channel and the data associated with each event takes the form of a string. The event channel propagates each event to the consumer, which simply displays the event data. This application is a simple example, but it illustrates a series of development tasks that apply to all OrbixTalk applications.

To develop an OrbixTalk application, you must implement the suppliers and consumers as normal ORB applications that communicate with the event channel through IDL interfaces. The applications specify which multicast protocol to use when binding to an event channel. The OrbixTalk IIOP Gateway implements the event channel. The IDL definitions for the CORBA Event Service are supplied with OrbixTalk.

This chapter describes the implementation of a supplier and consumer using Orbix for C++ as the development ORB. However, the OrbixTalk IIOP Gateway fully supports the CORBA Internet Inter-ORB Protocol (IIOP), so you may develop OrbixTalk applications using any IIOP-compatible ORB.

Developing an Untyped Push Supplier

As described in "Transfer of Untyped Events through an Event Channel" on page 43, a push supplier initiates the transfer of an event by pushing the event into an event channel. The event channel then takes responsibility for forwarding the event to each registered consumer.

This section describes how you can implement a push supplier as an Orbix application that communicates with a single event channel in an OrbixTalk server. This application acts as a client to several IDL interfaces implemented in the OrbixTalk event channel and acts as a server to the interface `PushSupplier`, which it implements.

These are the main programming steps in developing a push supplier:

1. Bind to an event channel in the IIOP Gateway.
2. Obtain a reference for a `ProxyPushConsumer` object from the event channel.

"Obtaining a ProxyPushConsumer from an Event Channel" on page 57 explains this step in detail.

3. Invoke the operation `connect_push_supplier()` on the `ProxyPushConsumer` object, to connect a `PushSupplier` implementation object to the event channel.

   "Connecting a PushSupplier Object to an Event Channel" on page 58 explains this step.

4. Invoke the `push()` operation on the `ProxyPushConsumer` object to initiate the transfer of each event.

   "Pushing Events to an Event Channel" on page 59 explains this step.

"The Push Supplier Application" on page 60 shows how these steps fit into a full Push supplier application.

Binding to an Event Channel

In OrbixTalk, every event channel has an associated event channel identifier which can be used to retrieve the channel's `EventChannel` object reference. Previously you could use the Orbix `_bind()` call to specify the channel identifier as the EventChannel object marker value, but `_bind()` is now deprecated for OrbixTalk Events Library. This means you can no longer bind to the event channel when your orbix application includes the OrbixTalk Events C++ Library. You must obtain a reference to the `OrbixTalkAdmin::OTChannelManager` object via the method `getOTChannelManager` on the new `OTChannelManagerAdmin` class. You can then get a reference to the event channel by invoking a method `get_event_channel` on the returned `OTChannelManager` object reference. (See "Using The Channel Manager to Retrieve Event Channels" on page 108 and "The OrbixTalkAdmin Module" on page 217 for more information.)

Obtaining a ProxyPushConsumer from an Event Channel

A push supplier needs to obtain a reference for a `ProxyPushConsumer` object in an event channel in order to transfer events to the channel for later distribution to consumers. The supplier transfers events by invoking the operation `push()` on the target `ProxyPushConsumer` object.

In order to obtain a ProxyPushConsumer object reference from an event channel, a supplier must implement the following programming steps:

1. Invoke the operation for_suppliers() on the EventChannel object, in order to obtain a SupplierAdmin object reference.
2. Invoke the operation obtain_push_consumer() on the SupplierAdmin object. This operation returns a ProxyPushConsumer object reference.

Connecting a PushSupplier Object to an Event Channel

When the supplier has retrieved the EventChannel object reference and used this to obtain a ProxyPushConsumer, the supplier needs to connect an implementation of the PushSupplier interface to the event channel. As described in "Registration of Suppliers and Consumers with an Event Channel" on page 38, this interface is defined as follows:

```
// IDL
module CosEventComm {
   ...

   interface PushSupplier {
      void disconnect_push_supplier ();
   };
};
```

The role of this interface is to allow the event channel to disconnect the supplier by invoking the operation disconnect_push_supplier(). This may happen if the event channel closes down.

In our example, the supplier implements the PushSupplier interface by defining the class PushSupplier_i, for example as follows:

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
...

class PushSupplier_i
   : public virtual CosEventComm::PushSupplierBOAImpl {

   public:
   unsigned char m_disconnected;
```

```
   PushSupplier_i () {
      m_disconnected = 0;
   }

   void disconnect_push_supplier (
      CORBA::Environment& env = CORBA::default_environment) {
      m_disconnected = 1;
   }
};
```

This class uses a simple flag mechanism to indicate the connection state of the supplier. The supplier connects an object of this type to an event channel by calling the operation `connect_push_supplier()` on the `ProxyPushConsumer` object.

Pushing Events to an Event Channel

The following code extract from the example supplier program is a simple demonstration of initiating the transfer of events:

```
// C++
while (!psImpl.m_disconnected) {
   CORBA::Any a;
   a <<= eventDataString;
   ppcVar->push (a);
}
```

In this example, the supplier repeatedly pushes an event to the event channel by calling the operation `push()` on a `ProxyPushConsumer` object. The supplier represents the event data using a simple string, but this is not necessary in general. The operation `push()` takes a parameter of type `any` for the event data, so you may represent this data using any IDL type.

Note that our supplier stops sending events only when it receives an incoming `disconnect_push_supplier()` operation call from the event channel. As an alternative, the supplier could explicitly disconnect from the event channel by invoking the operation `disconnect_push_consumer()` on the event channel `ProxyPushConsumer` object.

The Push Supplier Application

The three main programming steps in the development of push supplier applications have been described in detail.

The following source code illustrates how each of these steps fits in to the full push supplier application.

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
...

int main(int argc, char** argv) {
   char* eventDataString = "Hello World!";
   CosEventChannelAdmin::EventChannel_var ecVar;
   CosEventChannelAdmin::SupplierAdmin_var saVar;
   CosEventChannelAdmin::ProxyPushConsumer_var ppcVar;
   PushSupplier_i psImpl;
   char *serverHost;

   try {
      //
      // Step 1. Get a ProxyPushConsumer object reference.
      //

      // Obtain an event channel reference.
   try {
      ecVar = EventChannel::_bind ("otrmp//:ES",
         serverHost);
   }
   catch (...) {
      // Handle exception.
   ...
   }
      if (CORBA::is_nil (ecVar))
         return 1;

      // Obtain a supplier administration object.
      saVar = ecVar->for_suppliers ();

      // Obtain a proxy push consumer.
```

```
        ppcVar = saVar->obtain_push_consumer ();

        //
        // Step 2. Connect a push supplier implementation object.
        //
        ppcVar->connect_push_supplier (&psImpl);

        //
        // Step 3. Push events to the event channel.
        //
        while (!psImpl.m_disconnected) {
           CORBA::Any a;
           a <<= eventDataString;
           ppcVar->push (a);
           CORBA::Orbix.processNextEvent (1000);
        }

        // When finished, disconnect the consumer.
        ppcVar->disconnect_push_consumer();
     }
     catch (...) {
        // Handle exception
        ...
        return 1;
     }
     return 0;
}
```

Developing an Untyped Push Consumer

A push consumer receives events from an event channel, with no knowledge of the suppliers from which those events originated. An event channel propagates an event to a push consumer by invoking the operation `push()` on a `PushConsumer` implementation object in the consumer application. As such, the main functionality of a push consumer is associated with registering a `PushConsumer` object with an event channel and receiving incoming operation calls on that object.

To develop a push consumer application, you must implement the following steps:

1. Obtain a reference for a `ProxyPushSupplier` object from the event channel.

   "Obtaining a ProxyPushSupplier from an Event Channel" on page 63 explains this step.

2. Connect a `PushConsumer` implementation object to the event channel, by invoking the operation `connect_push_consumer()` on the `ProxyPushSupplier` object.

   "Connecting a PushConsumer Object to an Event Channel" on page 63 explains this step.

3. Monitor incoming operation calls.

   "Monitoring Incoming Operation Calls" on page 65 explains this step.

"The Push Consumer Application" on page 66 shows how these steps fit in to a full Push consumer application.

Obtaining a ProxyPushSupplier from an Event Channel

Each push consumer connected to an event channel receives every event raised by every supplier connected to the channel. However, consumers have no knowledge of the suppliers. Consumers simply connect to an object in the event channel which acts as a single source of events.

This object is responsible for storing a `PushConsumer` object reference for each connected consumer and invoking the `push()` operation on each of these references when a supplier transmits an event. The event channel object which stores consumer references is of type `ProxyPushSupplier`. The first task in developing a push consumer application is to obtain a reference to this object.

There are three stages in obtaining a `ProxyPushSupplier` object reference:

1. Obtain a reference to an `EventChannel` object in the event channel.
2. Invoke the operation `for_consumers()` on the `EventChannel` object to obtain a `ConsumerAdmin` object reference.
3. Invoke the operation `obtain_push_supplier()` on the `ConsumerAdmin` object. This operation returns a `ProxyPushSupplier` object reference.

You can implement the first of these steps in exactly the manner described for push supplier applications in "Obtaining a ProxyPushConsumer from an Event Channel" on page 57. The remaining steps involve normal operation invocations.

Connecting a PushConsumer Object to an Event Channel

When a consumer has obtained a reference to the `ProxyPushSupplier` object in an event channel, the next step is to register a `PushConsumer` implementation object with the `ProxyPushSupplier`. The event channel uses the `PushConsumer` object to propagate events to the consumer.

As described in "Registration of Suppliers and Consumers with an Event Channel" on page 38, the CORBA Event Service specification defines the interface `PushConsumer` as follows:

```
// IDL
module CosEventComm {
   interface PushConsumer {
      oneway void push (in any data) raises (Disconnected);
      void disconnect_push_consumer ();
   };
```

```
   ...
};
```

When an event arrives at an event channel, the channel `ProxyPushSupplier` object invokes the operation `push()` on each connected consumer, passing the event data as an `any` parameter. The `disconnect_push_consumer()` operation allows an event channel to disconnect a consumer, for example if the channel closes down.

Our consumer uses the following example implementation of this interface:

```
// C++
class PushConsumer_i
   : public virtual CosEventComm::PushConsumerBOAImpl {

   public:
   unsigned char m_disconnected;

   PushConsumer_i(){
      m_disconnected = 0;
   }

   virtual void disconnect_push_consumer (
      CORBA::Environment& env = CORBA::default_environment){
      m_disconnected = 1;
   }

   virtual void push (CORBA::Any& any,
      CORBA::Environment& env = CORBA::default_environment){
      char* msg;

      if (a >>= msg)
         cout << "Event received: event data = " << msg << endl;
      else
         cout <<
               "Event received with unexpected event data type."
              << endl;
   }
};
```

This class includes a trivial implementation of the `push()` operation, through which the consumer receives events. In normal OrbixTalk applications, this operation requires a more complex implementation which reacts appropriately to incoming events. The exact requirements for implementing the `push()` operation are application specific.

Monitoring Incoming Operation Calls

The main role of the consumer is to receive events from the event channel in the form of IDL operation calls. Consequently, the consumer must monitor and process any incoming calls. The example Orbix consumer application does this by repeatedly calling `processNextEvent()` on the `CORBA::Orbix` object, as follows:

```
// C++
while (!pcImpl.m_disconnected) {
   CORBA::Orbix.processNextEvent ();
}
```

The function `processNextEvent()` handles a single incoming operation call and then returns.

If the consumer receives an invocation on the operation `disconnect_push_consumer()`, then the implementation of this operation sets the value `pcImpl.m_disconnected` to one and breaks the consumer's event processing loop. Consequently, our consumer receives all events until the event channel explicitly forces it to disconnect.

As an alternative, the consumer could explicitly disconnect itself from the event channel when it no longer wishes to receive events. The consumer does this by invoking `disconnect_push_supplier()` on the event channel `ProxyPushSupplier` object.

The Push Consumer Application

The three main programming steps in the development of push consumer applications have been described in detail.

The following source code illustrates how each of these steps fits in to the full push consumer application.

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include <PushConsumer_i.h>

int main(int argc, char** argv) {
   CosEventChannelAdmin::EventChannel_var ecVar;
   CosEventChannelAdmin::ConsumerAdmin_var caVar;
   CosEventChannelAdmin::ProxyPushSupplier_var ppsVar;
   PushConsumer_i pcImpl;
   char *serverHost;

   try {
      //
      // Step 1. Get a ProxyPushSupplier object reference.
      //

      // Obtain an event channel reference.
      try {
      ecVar = EventChannel::_bind ("otrmp//:ES",
         serverHost);
      }
      catch (...) {
         // Handle exception.
   ...
      }
      if (CORBA::is_nil (ecVar))
         return 1;

      // Obtain a consumer administration object.
      caVar = ecVar->for_consumers ();

      // Obtain a proxy push supplier.
      ppsVar = caVar->obtain_push_supplier ();
```

```
        //
        // Step 2. Connect a push consumer implementation object.
        //
        ppsVar->connect_push_consumer (&pcImpl);

        //
        // Step 3. Monitor incoming operation calls.
        //
        while (!pcImpl.m_disconnected) {
           CORBA::Orbix.processNextEvent ();
        }

        // When finished, disconnect the supplier.
        ppsVar->disconnect_push_supplier();
     }
     catch (...) {
        // Handle exception.
        ...
        return 1;
     }
     return 0;
}
```

8

Programming with the Typed Push Model

To illustrate the use of the Push model to transmit typed events, this chapter develops a simple example.

This chapter describes how to develop an ORB application using the CORBA Event Service typed communications model that allows programmers to define an application-specific IDL interface. Callers can invoke operations defined on this interface to push events into an event channel. The parameters defined on these operations can specify the IDL method names and data types to be used to pass data on each event so the programmer is not restricted to passing data in an any.

As described in Chapter 6, "The Programming Interface to the Event Service", typed push model suppliers and consumers communicate through event channels supplied by the IIOP Gateway.

Overview of an Example Application

Consider a Stock Price application that reports the sales price of stock. The application that reports the sales price is a supplier of events. As well as reporting the price of stock, it may also generate events when the price of a particular stock exceeds a given threshold, when sales activity on the stock rises above a certain level, and so on.

Many different applications might be interested in receiving the events generated by the Stock Price application. These applications are consumers of events. Consumers might include stock brokers, insider trading watchdogs, government departments, and so on.

A suitable interface, supported by consumers of events for a Stock Price application, might be defined as follows:

```
// IDL
interface StockPrice
{
  oneway void quote(in float price);
};
```

Using this interface, a supplier application supplies events by invoking the `quote()` operation. The data associated with each event indicates the new price for the stock and takes the form of a `float`.

The simplified example that is described in this chapter consists of a typed push supplier and a push consumer, each of which connects to a single event channel. The supplier repeatedly pushes `StockPrice::quote()` events to the event channel. The event channel propagates each event to the consumer, which will simply display the event data. This application is simple, but it illustrates a series of development tasks that apply to all OrbixTalk applications using the typed push model.

Because event communication is unidirectional, operations defined on interface `StockPrice` must be `oneway` operations. Thus, the operation's parameters must be `in` parameters; the return value must be `void`; and the operation cannot have a `raises` clause.

When developing an OrbixTalk application, you must implement the suppliers and consumers as normal ORB applications that communicate with the event channel through IDL interfaces. OrbixTalk fully implements the event channel, which is created in the OrbixTalk IIOP Gateway. The IDL definitions for the CORBA Event Service are supplied with OrbixTalk.

This chapter examines the implementation of a supplier and consumer using Orbix for C++ as the development ORB. However, the OrbixTalk IIOP Gateway supports the CORBA Internet Inter-ORB Protocol (IIOP), so you may develop OrbixTalk applications using any IIOP-compatible ORB.

Developing a Typed Push Supplier

As described in Chapter 6, "The Programming Interface to the Event Service", a push supplier initiates the transfer of an event by pushing the event into an event channel. The event channel then takes responsibility for forwarding the event to each registered consumer.

This section describes how you can implement a typed push supplier as an Orbix application that communicates with a single event channel in an OrbixTalk server. This application acts as a client to several IDL interfaces implemented in the OrbixTalk event channel and acts as a server to the interface `PushSupplier`, which it implements.

There are four main programming steps in developing a typed push supplier:

1. Obtain a reference for a `TypedProxyPushConsumer` object from the event channel.

   "Obtaining a TypedProxyPushConsumer from an Event Channel" on page 72 explains this step in detail.

2. Invoke the operation `connect_push_supplier()` on the `TypedProxyPushConsumer` object, to connect a `PushSupplier` implementation object to the event channel.

   "Connecting a PushSupplier Object to an Event Channel" on page 73 explains this step.

3. Invoke the operation `get_typed_consumer()` on the `TypedProxyPushConsumer` object and narrow the returned `CORBA::Object` object reference to the appropriate application-specific type. "Obtaining a Typed Push Consumer from a ProxyPushConsumer" on page 74 explains this step.

4. Invoke an appropriate operation defined on the application-specific interface to initiate the transfer of each event.

   "Pushing Events to an Event Channel" on page 75 explains this step.

Obtaining a TypedProxyPushConsumer from an Event Channel

A typed push supplier needs to obtain a reference for a `TypedProxyPushConsumer` object in an event channel in order to transfer events to the channel for later distribution to consumers.

To obtain a `TypedProxyPushConsumer` object reference from an event channel, a supplier must implement the following programming steps:

1. Obtain a reference to a `TypedEventChannel` object in the event channel.
2. Invoke the operation `for_suppliers()` on the `TypedEventChannel` object, in order to obtain a `TypedSupplierAdmin` object reference.
3. Invoke the operation `obtain_typed_push_consumer()` on the `TypedSupplierAdmin` object, passing the name of the interface for which the typed consumer is required as a parameter to the operation. This operation returns a `TypedProxyPushConsumer` object reference.

These steps are defined in the CORBA Event Service specification and apply to all Event Service implementations.

In OrbixTalk, every event channel has an associated event channel identifier which can be used to retrieve the channel's `TypedEventChannel` object reference. When using the Orbix `_bind()` call, you can specify the channel identifier as the `TypedEventChannel` object marker value. For example:

```
// C++
TypedEventChannel_var channelVar;
char *serverHost;
...

try {
   channelVar = TypedEventChannel::
      _bind ("otrmp//:ES", serverHost);
}
catch (...) {
   // Handle exception.
   ...
}
```

Note that the server name for the OrbixTalk server is `ES`.

Connecting a PushSupplier Object to an Event Channel

When the supplier has retrieved the `TypedEventChannel` object reference and used this to obtain a `TypedProxyPushConsumer`, the supplier can connect an implementation of the `PushSupplier` interface to the event channel. As described in Chapter 6, "The Programming Interface to the Event Service", this interface is defined as follows:

```
// IDL
module CosEventComm {
   ...

   interface PushSupplier {
      void disconnect_push_supplier ();
   };
};
```

The role of this interface is to allow the event channel to disconnect the supplier by invoking the operation `disconnect_push_supplier()`. This may happen if the event channel closes down.

In our example, the supplier implements the `PushSupplier` interface by defining the class `TypedPushSupplier_i`, for example as follows:

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
...

class TypedPushSupplier_i
   : public virtual CosEventComm::PushSupplierBOAImpl {

   public:
   unsigned char m_disconnected;

   TypedPushSupplier_i () {
      m_disconnected = 0;
   }
```

```
   void disconnect_push_supplier (
      CORBA::Environment& env = CORBA::default_environment) {
      m_disconnected = 1;
   }
};
```

This class uses a simple flag mechanism to indicate the connection state of the supplier. The supplier connects an object of this type to an event channel by calling the operation `connect_push_supplier()` on the `TypedProxyPushConsumer` object.

Obtaining a Typed Push Consumer from a ProxyPushConsumer

To send typed events, the supplier must obtain a reference to an object in the event channel that supports the `StockPrice` interface. The supplier does this by invoking the operation `get_typed_consumer()` on the `TypedProxyPushConsumer` object it got from the event channel.

```
// C++
CORBA::Object_var objVar;
...
objVar = tppcVar->get_typed_consumer();
```

`get_typed_consumer()` returns an object reference of type `CORBA::Object`. Therefore, the supplier must narrow this object reference to a reference of type `StockPrice`.

```
// C++
if (StockPriceVar = StockPrice::_narrow (objVar)) {
   ...
else // call to _narrow() failed.
```

The supplier will use this object reference to push events to the event channel.

Pushing Events to an Event Channel

The following code extract from the example supplier program shows how the supplier initiates the transfer of events.

```
// C++
while (!tpsImpl.m_disconnected) {
   StockPriceVar->quote (24.60);
}
```

In this example, the supplier repeatedly pushes an event to the event channel by calling the operation `quote()` on a `StockPrice` object. The `StockPrice` object includes a `TypedPushConsumer` object that the supplier and consumer use to communicate typed events between them. The `quote()` operation takes one parameter of type `float` which contains the price of the stock item.

Note that our supplier stops sending events only when it receives an incoming `disconnect_push_supplier()` operation call from the event channel.

As an alternative, the supplier could explicitly disconnect from the event channel by invoking the operation `disconnect_push_consumer()` on the event channel `TypedProxyPushConsumer` object.

A Typed Push Supplier Application

The following source code implements a typed push supplier which supplies `StockPrice::quote()` events. It illustrates how the four programming steps described in detail in the preceding subsections fit in to a typed push supplier application.

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include "StockPrice_i.h"
...

int main(int argc, char** argv) {
   CosTypedEventChannelAdmin::TypedEventChannel_var tecVar;
   CosTypedEventChannelAdmin::TypedSupplierAdmin_var tsaVar;
   CosTypedEventChannelAdmin::TypedProxyPushConsumer_var tppcVar;
   CORBA::Object_var objVar;
   TypedPushSupplier_i tpsImpl;
   StockPrice_var StockPriceVar("IONAY", 24);
   char *serverHost;

   try {
      //
      // Step 1. Get a TypedProxyPushConsumer object reference.
      //

      // Obtain an typed event channel reference.
      try {
         tecVar = TypedEventChannel::
            _bind ("otrmp//:ES", serverHost);
      }
      catch (...) {
         // Handle exception.
   ...
      }
      if (CORBA::is_nil (tecVar))
         return 1;

      // Obtain a supplier administration object.
      tsaVar = tecVar->for_suppliers ();
```

```
        // Obtain a typed proxy push consumer.
        tppcVar = tsaVar->obtain_typed_push_consumer ("StockPrice");

        //
        // Step 2. Connect a push supplier implementation object.
        //
        tppcVar->connect_push_supplier (&tpsImpl);

        //
        // Step 3. Obtain a typed push consumer object reference.
        //
        objVar = tppcVar->get_typed_consumer();

        if (StockPriceVar = StockPrice::_narrow (objVar)) {
           //
           // Step 4. Push events to the event channel.
           //
           while (!tpsImpl.m_disconnected) {
              StockPriceVar->quote (24.60);
              CORBA::Orbix.processNextEvent (1000);
           }
        } else cout << "Attempt to _narrow() failed." << endl;

        // When finished, disconnect the consumer.
        tppcVar->disconnect_push_consumer();
     }
     catch (...) {
        // Handle exception
        ...
        return 1;
     }
     return 0;
  }
```

Developing a Typed Push Consumer

A typed push consumer receives events from an event channel, with no knowledge of the suppliers from which those events originated. The event channel, in turn, receives events from typed push suppliers in the form of operation invocations on the interface agreed between the suppliers and the event channel. An event channel propagates an event to a typed push consumer by invoking the operation on a `TypedPushConsumer` implementation object in the consumer application. As such, the main functionality of a typed push consumer is associated with registering a `TypedPushConsumer` object with an event channel and receiving incoming operation calls on that object.

To develop a typed push consumer application, you must implement the following steps:

1. Obtain a reference for a `ProxyPushSupplier` object from the event channel.

   "Obtaining a ProxyPushSupplier from an Event Channel" on page 79 explains this step.

2. Connect a `TypedPushConsumer` implementation object to the event channel, by invoking the operation `connect_push_consumer()` on the `ProxyPushSupplier` object, passing an object of type `TypedPushConsumer` as an operation parameter.

   "Connecting a TypedPushConsumer Object to an Event Channel" on page 79 explains this step.

3. Monitor incoming operation calls.

   "Monitoring Incoming Operation Calls" on page 82 explains this step.

"A Typed Push Consumer Application" on page 83 shows how these steps fit in to a full typed push consumer application.

Obtaining a ProxyPushSupplier from an Event Channel

Each typed push consumer connected to an event channel receives every event raised by every supplier connected to the channel. However, consumers have no knowledge of the suppliers. Consumers simply connect to an object in the event channel which acts as a single source of events.

The `ProxyPushSupplier` is responsible for storing the `TypedPushConsumer` object reference for a connected consumer and propagating the operation invocation it receives when a supplier transmits an event. The first task in developing a push consumer application is to obtain a reference to this object.

There are three stages in obtaining a `ProxyPushSupplier` object reference:

1. Obtain a reference to a `TypedEventChannel` object in the event channel.
2. Invoke the operation `for_consumers()` on the `TypedEventChannel` object, in order to obtain a `TypedConsumerAdmin` object reference.
3. Invoke the operation `obtain_typed_push_supplier()` on the `TypedConsumerAdmin` object and pass the name of the interface agreed between the event channel and the typed consumer as a parameter. This operation returns a `ProxyPushSupplier` object reference.

You may implement the first of these steps in exactly the manner described for typed push supplier applications in "Obtaining a TypedProxyPushConsumer from an Event Channel" on page 72. The remaining steps involve normal operation invocations.

Connecting a TypedPushConsumer Object to an Event Channel

When a typed consumer has obtained a reference to the `ProxyPushSupplier` object in an event channel, the next step is to register a `TypedPushConsumer` implementation object with the `ProxyPushSupplier`. The event channel checks that the Push consumer object registered can be narrowed to a `TypedPushConsumer`. After a brief time the event channel invokes `get_typed_consumer()` on the `TypedPushConsumer` object reference.

As described in Chapter 6, "The Programming Interface to the Event Service", the CORBA Event Service specification defines the interface TypedPushConsumer as follows:

```
// IDL
module CosTypedEventComm {
   interface TypedPushConsumer : CosEventComm::PushConsumer {
      Object get_typed_consumer ();
   };
   ...
};
```

When an event arrives at an event channel in the form of an invocation on any of the operations defined in the interface, the channel invokes the same operation on each connected consumer. The disconnect_push_consumer() operation allows an event channel to disconnect a consumer, for example, if the channel closes down.

This is a example implementation of the agreed interface, StockPrice. When an object of this class is created, it in turn creates an object of class TypedPushConsumer_i. When the quote operation occurs, the stockprice object carries out an operation on the TypedPushConsumer:

```
//C++
class TypedPushConsumer_i;

class StockPrice_i : public StockPriceBOAImpl
{
public:
  StockPrice_i (TypedPushConsumer_i*  pCons);

  ~StockPrice_i();

  virtual void quote
  (
    CORBA::Float price,
    CORBA::Environment &
  ) throw (CORBA::SystemException);

private:
  TypedPushConsumer_i*    m_pCons;
};
```

This is an example implementation of the TypedPushConsumer interface:

```
//C++
class TypedPushConsumer_i : public
CosTypedEventComm::TypedPushConsumerBOAImpl
{
   CORBA(Boolean) m_bConnected;
   StockPriceBOAImpl* m_pStockPrice;

   TypedPushConsumer_i(unsigned int numMsgsToConsume ) :
      m_bConnected(FALSE)
   {
      m_pStockPrice = new StockPrice_i(this);
   }

   ~TypedPushConsumer_i()
   {
      CORBA::release(m_pStockPrice);
   }

// We come here when there is an incoming typed push event from
some supplier
// via the StockPrice_i class quote method
//
   void do_quote(float price )
   {
      m_quoteCount++;
      cout << "[TypedPushConsumer: got a typed push (quote " <<
            m_quoteCount<< " of : " << price << "]" << endl;
   }

   void push( const CORBA(any)&   a,CORBA(Environment)&){
   {
      throw CORBA::NO_IMPLEMENT;
   }

...
}
```

Because TypedPushConsumer inherits from PushConsumer, it must provide an implementation of the push() operation defined on interface PushConsumer. In this example, class StockPrice_i provides a null implementation of push(), which simply raises the standard CORBA exception CORBA::NO_IMPLEMENT. This restricts suppliers to using typed communication with this consumer.

Alternatively, class TypedPushconsumer_i could implement push() so that the consumer receives untyped as well as typed events.

Monitoring Incoming Operation Calls

The main role of the typed consumer is to receive events from the event channel in the form of IDL operation calls. Consequently, the consumer must monitor and process any incoming calls. The example Orbix consumer application does this by repeatedly calling processNextEvent() on the CORBA::Orbix object, as follows:

```
// C++
while (!StockPriceImpl.m_disconnected) {
   CORBA::Orbix.processNextEvent ();
}
```

The function processNextEvent() handles a single incoming operation call and then returns.

If the consumer receives an invocation on the operation disconnect_push_consumer(), then the implementation of this operation sets the value TypedPushConsumer_i.m_disconnected to one and breaks the consumer's event processing loop. Consequently, our consumer receives all events until the event channel explicitly forces it to disconnect.

As an alternative, the consumer could explicitly disconnect itself from the event channel when it no longer wishes to receive events. The consumer does this by invoking disconnect_push_supplier() on the event channel ProxyPushSupplier object.

A Typed Push Consumer Application

The three main programming steps in the development of a typed push consumer applications have been described in detail.

The following source code illustrates how each of these steps fits in to the full typed push supplier application. The application obtains a typed proxy push supplier for the interface `StockPrice` and then waits for events.

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include "TypedPushConsumer_i.h"
#include "StockPrice_i.h"

int main(int argc, char** argv) {
   CosTypedEventChannelAdmin::TypedEventChannel_var tecVar;
   CosTypedEventChannelAdmin::TypedConsumerAdmin_var tcaVar;
   CosTypedEventChannelAdmin::ProxyPushSupplier_var ppsVar;
   TypedPushConsumer_i TypedPushConsumerImpl;
   char *serverHost

   try {
      //
      // Step 1. Get a ProxyPushSupplier object reference.
      //

      // Obtain a typed event channel reference.
      try {
         tecVar = TypedEventChannel::_bind (
            "otrmp//:ES", serverHost);
      }
      catch (...) {
         // Handle exception.
   ...
      }
      if (CORBA::is_nil (tecVar))
         return 1;

      // Obtain a typed consumer administration object.
      tcaVar = tecVar->for_consumers ();
```

```
        // Obtain a typed proxy push supplier for
        // the interface StockPrice.
        ppsVar = tcaVar->obtain_typed_push_supplier ("StockPrice");

        //
        // Step 2. Connect a typed push consumer
        // implementation object.
        //
        ppsVar->connect_push_consumer (&TypedPushConsumerImpl);

        //
        // Step 3. Monitor incoming operation calls.
        //
        while (!TypedPushConsumerImpl.m_disconnected) {
           CORBA::Orbix.processNextEvent ();
        }

        // When finished, disconnect the supplier.
        tppsVar->disconnect_push_supplier();
     }
     catch (...) {
        // Handle exception.
        ...
        return 1;
     }
     return 0;
}
```

9

Programming with the Untyped Pull Model

To illustrate the Pull model to transfer untyped events, this chapter develops a simple application.

As described in Chapter 5, "The CORBA Event Service", OrbixTalk allows you to develop Object Request Broker (ORB) applications that communicate using the CORBA Event Service communications model. From a programmer's perspective, the event channel is the key element of a CORBA Event Service application.

This chapter describes an example ORB application that illustrates how you can use OrbixTalk to develop pull model suppliers and consumers that communicate untyped events through event channels.

Overview of an Example Application

The example described in this chapter consists of a pull supplier and a pull consumer, each of which connects to a single event channel. The consumer repeatedly pulls an event from the event channel. The data associated with each event takes the form of a string, and the consumer simply displays the data as it receives it. The event channel, in turn, pulls the data from a pull supplier. This application is straightforward, but it illustrates a series of development tasks that apply to all OrbixTalk applications.

When developing an OrbixTalk application, you must implement the suppliers and consumers as normal ORB applications that communicate with the event channel through IDL interfaces. The OrbixTalk IIOP Gateway implements the event channel. The IDL definitions for the CORBA Event Service are supplied with OrbixTalk.

This chapter examines the implementation of a supplier and consumer using Orbix for C++ as the development ORB. However, the OrbixTalk server fully supports the CORBA IIOP, so you may develop OrbixTalk applications using any IIOP-compatible ORB.

Developing an Untyped Pull Consumer

As described in "Transfer of Typed Events Through an Event Channel" on page 49, a pull consumer initiates the transfer of an event by requesting the event from the event channel. The event channel, if it does not already have an event to meet the request, requests an event from each registered supplier and then passes an event to the pull consumer. A pull consumer may poll for an event if it does not want to block while waiting for an event to become available.

To develop a pull consumer application, you must implement the following steps:

1. Obtain a `ProxyPullSupplier` object from the event channel.
2. Invoke the operation `connect_pull_consumer()` on the `ProxyPullSupplier` object, to connect a `PullConsumer` implementation object to the event channel.
3. Invoke `try_pull()` operations on the `ProxyPullSupplier` object to initiate the transfer of each event. (As an alternative you can also use the `pull()` operation. `try_pull()` is preferred however.)

Obtaining a ProxyPullSupplier from an Event Channel

A pull consumer connected to an event channel receives an event only when it explicitly requests one. The consumer has no knowledge of the suppliers; it simply connects to an object in the event channel that acts as a single source of events.

This object is responsible for storing a `PullSupplier` object reference for each connected supplier and invoking a `try_pull()` or `pull()` operation on each of these object references when a consumer requests an event using `try_pull()` or `pull()` respectively. The event channel object which stores supplier references is of type `ProxyPullSupplier`. The first task in developing a push consumer application is to obtain a reference to this object.

As illustrated in our example pull consumer application, a pull consumer obtains a reference to a `ProxyPullSupplier` by implementing the following steps:

1. Obtain a reference to an `EventChannel` object in the event channel.
2. Invoke the operation `for_consumers()` on the `EventChannel` object in order to obtain a `ConsumerAdmin` object.
3. Invoke the operation `obtain_pull_supplier()` on the `ConsumerAdmin` object. This operation returns a `ProxyPullSupplier` object reference.

You may implement the first of these steps exactly as described for push supplier applications in "Obtaining a ProxyPushConsumer from an Event Channel" on page 57. The remaining steps involve normal operation invocations.

Connecting a PullConsumer Object to an Event Channel

When the consumer has obtained a reference to a `ProxyPullSupplier` object from the event channel, it needs to connect an implementation of the `PullConsumer` interface to the event channel. As described in "The Pull Model for Untyped Events" on page 41, this interface is defined as follows:

```
// IDL
module CosEventComm {
   ...

   interface PullConsumer {
      void disconnect_pull_consumer ();
   };
};
```

The purpose of this interface is to allow the event channel to disconnect the PullConsumer by invoking the operation disconnect_pull_consumer(). This may be necessary if the event channel closes down.

In our example, the consumer application implements the PullConsumer interface by defining the class PullConsumer_i and implementing it as follows:

```
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
...

class PullConsumer_i
   : public virtual CosEventComm::PullConsumerBOAImpl {

   public:
   CORBA::Boolean m_disconnected;

   PullConsumer_i () {
      m_disconnected = 0;
   }

   void disconnect_pull_consumer (
      CORBA::Environment& env = CORBA::default_environment) {
      m_disconnected = 1;
   }
};
```

Class PullConsumer_i uses a simple flag mechanism to indicate the connection state of the consumer. The consumer connects an object of this type to an event channel by calling the operation connect_pull_consumer() on the ProxyPullSupplier object.

Pulling Events from an Event Channel

The following code extract from the example consumer program shows a simple way of initiating the transfer of events using the `try_pull()` operation:

```
CORBA::Boolean got_event;
...
while (!pcImpl.m_disconnected) {
   event = ppsVar->try_pull(got_event);
   if (got_event) {
      if (*event >>= eventDataString) {
         cout << eventDataString << endl;
         delete [] eventDataString;
         delete event;
      } else {
         cout << "Error: Pulled bad data" << endl;
      }
   } else {
      cout << "Event channel did not supply event" << endl;
   }
   CORBA::Orbix.processNextEvent (1000);
}
```

In this example, the consumer repeatedly pulls an event from the event channel using the `try_pull()` operation on a `ProxyPullSupplier` object in the channel. In this example, the event supplied in the any return value of the `try_pull()` operation is a string; in general, the type contained in this any is application dependent.

The `try_pull()` operation pulls events without blocking. The `pull()` operation causes the consumer application to block until a event is supplied by the channel. If you are using a multi-thread safe ORB such as Orbix or OrbixWeb, you could also create an application thread dedicated to pulling events from the channel without blocking the consumer application. The following code extract illustrates the use of `pull()`:

```
// C++
while (!pcImpl.m_disconnected) {
   event = ppsVar->pull();
   if (*event >>= eventDataString) {
      cout << eventDataString << endl;
      delete [] eventDataString;
      delete event;
```

```
   } else {
      cout << "Error: Pulled bad data" << endl;
   }
   CORBA::Orbix.processNextEvent (1000);
}
```

The consumer stops pulling events only when it receives an incoming `disconnect_pull_consumer()` operation call from the event channel. Alternatively, the consumer could explicitly disconnect from the event channel by invoking the operation `disconnect_pull_supplier()` on the `ProxyPullSupplier` object in the event channel.

An Untyped Pull Consumer Application

The following source code illustrates the implementation of a simple pull consumer that pulls events using the `try_pull()` operation:

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include <PullConsumer_i.h>
...
int main(int argc, char** argv) {
   CosEventChannelAdmin::EventChannel_var ecVar;
   CosEventChannelAdmin::ConsumerAdmin_var caVar;
   CosEventChannelAdmin::ProxyPullSupplier_var ppsVar;
   CORBA::Any* event;
   char* eventDataString;
   PullConsumer_i pcImpl;
   CORBA::Boolean got_event;
   char *serverHost;


   try
      //
      // Step 1. Get a ProxyPullSupplier object reference.
      //

      // Obtain an event channel reference.
   try {
      ecVar = EventChannel::_bind ("otrmp//:ES",
```

```
      serverHost);
}
catch (...) {
   // Handle exception.
...
}
   if (CORBA::is_nil (ecVar))
      return 1;

   // Obtain a consumer administration object.
   caVar = ecVar->for_consumers ();

   // Obtain a proxy pull supplier.
   ppsVar = saVar->obtain_pull_supplier ();

   //
   // Step 2. Connect a pull consumer implementation object.
   //
   ppsVar->connect_pull_consumer (&pcImpl);

   //
   // Step 3. Pull events from the event channel.
   //
   while (!pcImpl.m_disconnected) {
      event = ppsVar->try_pull(got_event);
      if (got_event){
         if(*event >>= eventDataString){
            cout << eventDataString << endl;
            delete[] eventDataString;
            delete event;
         } else {
            cout << "Error: Pulled bad data" << endl;
      else {
         cout << "Event channel did not supply event" << endl;

      }
      CORBA::Orbix.processNextEvent (1000);
   }
}
catch (...) {
   // Handle exception.
   ...
   return 1;
```

```
    }
    return 0;
}
```

"Obtaining a ProxyPullSupplier from an Event Channel" on page 87, "Connecting a PullConsumer Object to an Event Channel" on page 87, and "Pulling Events from an Event Channel" on page 89 explain the details of each step in the implementation of a `PullConsumer` with reference to this source code.

Developing an Untyped Pull Supplier

A pull supplier supplies events on request to an event channel and has no knowledge of the consumers to which these events will be propagated. The event channel requests an event from a pull supplier in order to fulfil a request for an event by a pull consumer. An event channel requests an event by invoking the `pull()` or `try_pull()` operations on a `PullSupplier` object in the supplier application. A supplier application, therefore, must register a `PullSupplier` object with an event channel and receive incoming operation calls on that object.

To develop a pull supplier application, you must implement the following steps:

1. Obtain a reference for a `ProxyPullConsumer` in the event channel.
2. Connect a `PullSupplier` implementation object to the event channel by invoking the operation `connect_pull_supplier()` on the `ProxyPullConsumer` object.
3. Monitor incoming operation calls.

Obtaining a ProxyPullConsumer from an Event Channel

When a pull consumer requests an event, the event channel to which it is connected in turn requests an event from each connected pull supplier if it does not already have an event stored in the channel. The suppliers have no knowledge of the consumers requesting events; they simply connect to an object in the event channel.

This object is responsible for storing a `PullSupplier` object reference for each connected supplier, and invoking the `pull()` or `try_pull()` operation on each of these references when a consumer requests an event. The event channel

object which stores supplier references is of type ProxyPullConsumer. The first task in developing a pull supplier application is to obtain a reference to this object.

As illustrated in our example supplier source code, this reference is obtained by implementing the following steps:

1. Obtain a reference to an EventChannel object in the event channel.
2. Invoke the operation for_suppliers() on the EventChannel object in order to obtain a SupplierAdmin object.
3. Invoke the operation obtain_pull_consumer() on the SupplierAdmin object. This operation returns a ProxyPullConsumer object reference.

You may implement the first of these steps exactly as described for push supplier applications in "Obtaining a ProxyPushConsumer from an Event Channel" on page 57. The remaining steps involve normal operation invocations.

Connecting a PullSupplier Object to an Event Channel

When a supplier has obtained a reference to a ProxyPullConsumer object in an event channel, the next step is to register a PullSupplier implementation object with the ProxyPullConsumer.

As described in "The Pull Model for Untyped Events" on page 41, the CORBA Event Service specification defines the interface PullSupplier as follows:

```
// IDL
module CosEventComm {
   interface PullSupplier {
      any pull () raises (Disconnected);
      any try_pull (out boolean has_event) raises (Disconnected);
      void disconnect_pull_supplier();
   };
   ...
};
```

When a request for an event arrives at an event channel in the form of a pull() or try_pull() operation from a pull consumer, the channel ProxyPullConsumer object invokes a corresponding pull() or try_pull() operation on each connected supplier.

The disconnect_pull_supplier() operation allows the event channel to disconnect a supplier, for example, if the event channel closes down.

Our example supplier implements this interface as follows:

```
// C++
class PullSupplier_i :
   public virtual CosEventComm::PullSupplierBOAImpl {

   protected:
   unsigned char m_generate_event;

   public:
   CORBA::Boolean m_disconnected;

   PullSupplier_i () {
      m_disconnected = 0;
      m_have_event = 0;
   }

   virtual void disconnect_pull_supplier (
      CORBA::Environment& env = CORBA::default_environment){
      m_disconnected = 1;
   }

   virtual CORBA::Any* pull (
      CORBA::Environment& env = CORBA::default_environment){
      CORBA::Any a;
      char* eventDataString = "Hello World!";
      if (!m_disconnected) {
         a <<= eventDataString;
         return a;
      } else {
         throw CosEventComm::Disconnected;
         return 0;
      }
   }

   virtual CORBA::Any* try_pull (
      CORBA::Boolean& has_event,
      CORBA::Environment& env = CORBA::default_environment){
         // This trivial implementation of try_pull()
         // supplies an event on every alternate call.
         CORBA::Any a;
         char* eventDataString = "Hello World!";
         if (!m_disconnected) {
```

```
            if (m_generate_event) {
               a <<= eventDataString;
               m_has_event = 1;
               m_generate_event = 0;
               return a;
            }
            else {
               m_has_event = 0;
               m_generate_event = 1;
            }
         }
      } else {
         throw CosEventComm::Disconnected;
         return 0;
      }
   }
};
```

This class includes trivial implementations of the `pull()` and `try_pull()` operations which deal with requests for events. The exact requirements for implementing these operations are application specific; a real OrbixTalk application would probably require more complex implementations.

Monitoring Incoming Operation Calls

A pull supplier application receives requests for events from an event channel in the form of `pull()` and `try_pull()` operation calls; the event channel may also disconnect the supplier by invoking the operation `disconnect_pull_supplier()`. The supplier must, therefore, monitor and process incoming operation calls. The example pull supplier application does this by repeatedly calling `processNextEvent()` on the `CORBA::Orbix` object, as follows:

```
while (!psImpl.m_disconnected) {
   CORBA::Orbix.processNextEvent (1000);
}
```

The function `processNextEvent()` handles a single incoming operation call and then returns. This example uses a timeout value of 1000 milliseconds, but any finite value would be appropriate.

If the supplier receives a `disconnect_pull_supplier()` operation invocation, then the implementation of this operation sets the value `psImpl.m_disconnected` to one and breaks the supplier's event processing loop. In this way, our supplier receives operation invocations until the event channel explicitly asks it to disconnect. A supplier could explicitly disconnect itself from the event channel when it no longer wants to supply events, by invoking the operation `disconnect_pull_consumer()` on the event channel `ProxyPullConsumer` object.

An Untyped Pull Supplier Application

The following code implements an example pull supplier:

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include <PullSupplier_i.h>

int main(int argc, char** argv) {
   char *eventChannelName = "Channel_1"
   CosEventChannelAdmin::EventChannel_var ecVar;
   CosEventChannelAdmin::SupplierAdmin_var saVar;
   CosEventChannelAdmin::ProxyPullConsumer_var ppcVar;
   PullSupplier_i psImpl;

   try {
      //
      // Step 1. Get a ProxyPullConsumer object reference.
      //

      // Obtain an event channel reference.
   try {
      ecVar = EventChannel::_bind ("otrmp//:ES",
         serverHost);
   }
   catch (...) {
      // Handle exception.
   ...
   }
      if (CORBA::is_nil (ecVar))
```

```
            return 1;

         // Obtain a supplier administration object.
         saVar = ecVar->for_suppliers ();

         // Obtain a proxy pull consumer.
         ppcVar = saVar->obtain_pull_consumer ();

         //
         // Step 2. Connect a pull supplier implementation object.
         //
         ppcVar->connect_pull_supplier (&psImpl);

         //
         // Step 3. Monitor incoming operation calls.
         //
         while (!psImpl.m_disconnected) {
            CORBA::Orbix.processNextEvent (1000);
         }
      }
      catch (...) {
         // Handle exception.
         ...
         return 1;
      }
      return 0;
   }
```

10

The OrbixTalk Events Library

The events library enables C++ applications to send multicast messages directly without the IIOP Gateway.

The OrbixTalk events library is an implementation of the CORBA Events Service specification. It provides a C++ library that can be included in any Orbix application so that it can use multicast functionality with the Event Service IDL, but without having to bind to an event channel provided by the IIOP Gateway.

The library provides each supplier or consumer with its own smart proxy. Supplier and consumer smart proxies can cooperate to produce an effective event channel, but multicast communication occurs directly. This implementation means that you can develop C++ suppler and consumer applications in the same way as those that use the IIOP Gateway as demonstrated in Chapter 7, "Programming with the Untyped Push Model" and the following chapters. The only differences are that:

- You must include the C++ library header files rather than IDL generated header files.
- You must bind to the event channel using different channel identifiers, and no server name. Alternatively, the `OrbixTalkAdmin::ChannelManager` interface can be used to manage event channels.

The C++ Library Header Files

There are two header files associated with the C++ library. These two files contain definitions and static declarations of smart proxy factories:

`coseventsadmin.h`	This provides smart proxy factories for `CosEventChannelAdmin::EventChannel` and `CosTypedEventChannelAdmin::TypedEventChannel`.
`orbixtalkadmin.h`	This provides smart proxy factories for `OrbixTalkAdmin::ChannelManager`.

These smart proxy factories allocate a smart proxy object within the application's address space when you call `_bind(char* channelname, "")` on one of these classes.

Note that the equivalent header files used when binding to event channels provided by the IIOP Gateway are named `coseventsadmin.hh` and `orbixtalkadmin.hh`.

The library allows multiple event channels within one application. It is designed to allow one or more suppliers to supply events to each event channel, and one or more consumers to consumer events from each event channel.

Event Channel Identifiers

Each event channel can use reliable multicast protocol (`rmp`), store and forward protocol (`sfp`), or basic multicast (`mcp`). An non-multicast event channel can be determined from the prefix of the event channel's name. The following prefixes are recognized:

`otrmp//`	Reliable multicast.
`otsfp//`	Store and forward multicast.
`otmcp//`	Raw multicast (fire and forget).

If no prefix is specified, an invalid name exception (`OrbixTalkAdmin::InvalidName`) is thrown. This differs from the IIOP Gateway.

For example, if you want to bind to an event channel using the C++ library, you can use this call:

```
_bind("otrmp//","")
```

Store and Forward Multicast

Store and Forward replay can be controlled using the `OrbixTalkAdmin::ChannelManager` interface.

The Events Library and The OrbixTalk Daemon

At start-up the application attempts to contact the OrbixTalk daemon (otd). When an event channel is created the application contacts the OrbixTalk daemon to obtain OrbixTalk topic information for that event channel. Queries on the OrbixTalk daemon from other applications and gateways for the same channel name are provided with the same topic information.

Non-Multicast Event Channels

These are not available in the OrbixTalk events library.

11

OrbixTalk IIOP Gateway

The OrbixTalk IIOP Gateway is a CORBA server. It provides event channels that allow any IIOP client such as OrbixWeb to communicate using multicast functionality.

The gateway provides multiple event channels including both typed and untyped channels. You can develop any IIOP conformant client as a supplier or consumer which can connect to a channel. The gateway allows one or more suppliers to supply events to each event channel, and one or more consumers to receive events from each event channel.

You can develop suppliers and consumers in the way described in Chapter 7, "Programming with the Untyped Push Model" and the following chapters. All you must do to use an event channel provided by the Gateway is:

- Include the header files `coseventsadmin.hh` and `orbixtalkadmin.hh` where necessary.
- Use the appropriate channel identifier when binding to an event channel.

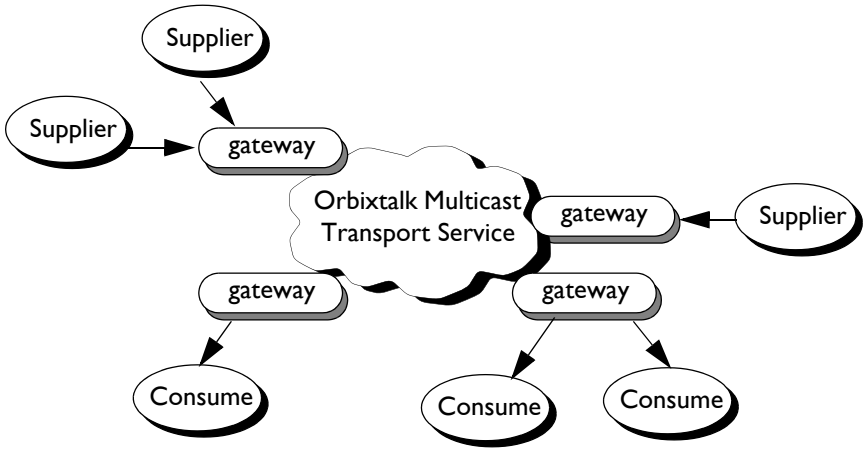

Figure 11.1: *OrbixTalk IIOP Gateway*

Event Channel Identifiers

Each event channel can use reliable multicast protocol (`rmp`), store and forward protocol (`sfp`), or basic multicast (`mcp`). An non-multicast event channel may specified determined from the prefix of the event channel's name. The following prefixes are recognized:

`otrmp//`	Reliable multicast.
`otsfp//`	Store and forward multicast.
`otmcp//`	Raw multicast (fire and forget).
`(No Prefix)`	Non-multicast. (Use none of the above.)

Store and Forward Multicast

Although the gateway supports more than one consumer, the full `sfp` functionality is only available to one consumer at a time. The application name is used by `sfp` requesting the replay of events from the message store. The application name can be set once (see "Using The Channel Manager to Retrieve Event Channels" on page 108).

The IIOP Gateway and The OrbixTalk Demon

At start-up the event server attempts to contact the orbixTalk daemon (`otd`). If it cannot do so then it will not become available on the network.

When an event channel is created the event server checks to see if it is a multicast event channel. If it is the event server contacts the OrbixTalk daemon to obtain OrbixTalk topic information for that event channel. Queries on the OrbixTalk daemon from other event servers for the same channel name are provided with the same topic information.

The event server does not need to contact the OrbixTalk daemon for non-multicast channels. It simply provides an internal mechanism for the event channel. This is the most significant difference between an event server licensed for Orbix, and one with an OrbixTalk license. If you are using Orbix only, multicast channels are not allowed, and the event server does not need to contact the OrbixTalk daemon, either at start-up or during creation of an event channel.

Non-Multicast Event Channels

When an event is supplied to a non-multicast event channel the event server buffers it internally as though it was to be multicast. Instead of transmitting the event, it is passed to the receive thread as though it had been received from the network. There may not be a receive thread if the event server has no connected consumers on that event channel. In that case the event is discarded.

The IIOP Gateway Command Lines

The gateway's command line allow you to configure how it is launched. It takes the following form:

```
otgateway [option | channel id]
```

Where the option is one of the command-line parameters listed in Table 11.2.

Option	Effect
untyped_channels	Subsequent channel names are treated as untyped. This is the default.
typed_channels	Subsequent channel names are treated as typed.
not_orbix_server	Do not call CORBA::Orbix.setServerName() or CORBA::Orbix.impl_is_ready().
server_name	The name passed to CORBA::Orbix.setServerName() and CORBA::Orbix.impl_is_ready(). Default is "ES".
srv_timeout	Millisecond time passed to single call to processEvents(). Default is INFINITE.
default_tx_timeout	Millisecond time passed to CORBA::Orbix.defaultTxTimeout(). Default is INFINITE.
use_transient_port	CORBA::Orbix.useTransientPort(1) is called.
set_diagnostics	Value of 0, 1, 2 or 3 passed to CORBA::Orbix.setDiagnostics(). Default is 0.
robust	Invocations to destroy on typed or untyped events channel will fail if any Proxies exist.

Table: 11.2: *Gateway Command-line Options*

Option	Effect
always_try_pull_on_suppliers	If a PullConsumer calls pull() on a ProxyPullSupplier then, by default, all ProxyPullConsumers call pull() on their connected PullSuppliers. If this parameter is set then ProxyPullConsumers call try_pull() on their connected PullSuppliers. Note that if a PullConsumer calls try_pull() on a ProxyPullSupplier then all ProxyPullConsumers will always call try_pull() on their connected PullSuppliers.
pull_prod_interval	When a PullConsumer calls pull() on a ProxyPullSupplier, this parameter sets the interval between: New ProxyPullConsumers calling pull() on their connected PullSuppliers. All ProxyPullConsumers calling try_pull() on their connected PullSuppliers.
try_pull_duration	When a PullConsumer calls try_pull() on a ProxyPullSupplier then this is the duration that try_pull() blocks, awaiting an event, before returning with has_event set to FALSE.
-nonames	Do not place the name OrbixTalkAdminChannelManager in root context of Name Service. (Default places it in root context if Name Service is running.)
I or i	OrbixTalkAdmin IOR is written to a file OrbixTalkAdmin.ref.
v	Version information and opal version information/configuration.

Table: 11.2: *Gateway Command-line Options*

Option	Effect
?	Usage information.
D	Dump current configuration settings.

Table: 11.2: *Gateway Command-line Options*

Using The Channel Manager to Retrieve Event Channels

The IIOP Gateway supports the standard COSEvents IDL interface as defined in the OMG CORBA specification. In addition, it provides an OrbixTalkAdmin module which enables you to obtain references to event channels, and specify the persistent application name for the gateway.

```
#include "coseventsadmin.idl"
module OrbixTalkAdmin
{
   typedef string ChannelName;
   exception InvalidName{ };
   exception InvalidOption{ };
   exception AlreadySet{ };

  // Replay Type
  //
  //REPLAY_NONE - Do not replay messages (Default)
  // REPLAY_ALL      - Replay all messages
  // REPLAY_RELATIVE - Replay n messages relative
to most recent
  // REPLAY_ABSOLUTE - Replay messages starting
from n
// REPLAY_USE_EXISTING - Use the existing replay
mechanism for the channel
  //
  typedef unsigned short ReplayType;
  typedef unsigned long  ReplayValue;

const ReplayType REPLAY_NONE = 0;
const ReplayType REPLAY_ALL = REPLAY_NONE + 1;
```

```
const ReplayType REPLAY_RELATIVE = REPLAY_NONE +
2;
const ReplayType REPLAY_ABSOLUTE = REPLAY_NONE +
3;
const ReplayType REPLAY_USE_EXISTING = REPLAY_NONE
+ 6;

  interface OTChannelManager
  {
    CosEventChannelAdmin::EventChannel
get_event_channel
    (
      in    ChannelName  channel_name,
      inout ReplayType   type,
      inout ReplayValue  value
    )       raises (InvalidName,InvalidOption);

    CosTypedEventChannelAdmin::TypedEventChannel
get_typed_event_channel
    (
      in    ChannelName  channel_name,
      inout ReplayType   type,
      inout ReplayValue  value
    )       raises (InvalidName,InvalidOption);

    // required for SFP
void set_app_name
    (
      in string name
    ) raises (InvalidName, AlreadySet);
  };
};
```

Note: To replay the last message sent, set `ReplayRelative` to 1.

In order to use an event channel, you need to bind to the `OTChannelManager` object then call `get_event_channel()`:

```
managerVar = OrbixTalkAdmin::OTChannelManager::_bind(":ES");
channelVar = managerVar->get_event_channel
   ("otrmp//Events",OrbixTalkAdmin::ReplayAll,0);
```

The replay parameters are ignored for topics not using the Store and Forward Protocol (otsfp).

The executable file for the gateway is otgateway. You do not need to specify any parameters when you use the otgateway file with an Orbix daemon. Register the gateway with the Orbix daemon as a per-client server to ensure that each client process has access to its own gateway:

```
putit -per-client-pid ES <path>/otgateway -srv_timeout n
```

It is highly recommended that you register the gateway in like this, to avoid the otgateway becoming a bottleneck. The otgateway must be registered using the per-client-pid activation mode for otsfp listener applications. This is because each listener using the otsfp protocol has its own state log file. This statelog is used by OrbixTalk to determine the last message that was successfully received if a listener exits and restarts for any reason. The missed messages can then be replayed for the listener. As there is only one state log per process it is necessary for the orbixd to launch a separate gateway process for each client.

During start-up the gateway attempts to call resolve_initial_references("NameService"). If this is successful then the OrbixEventsAdmin::ChannelManager and OrbixTalkAdmin::OTChannelManager object references are placed in the root CosNaming::NamingContext.

Part III

Managing OrbixTalk

12

Building and Running OrbixTalk Applications

This chapter lists the libraries required to build applications on different platforms using a multi-threaded environment.

Overview

Before running an application, ensure that one instance of the OrbixTalk Directory Enquiries daemon (`otd` or `otdsm`) is running on each subnet.

If the application uses the OrbixTalk MessageStore, ensure that one instance of the OrbixTalk Directory Enquiries daemon (`otd` or `otdsm`) and one instance of the OrbixTalk MessageStore daemon (`otmsd`) are running on each subnet. The OrbixTalk Directory Enquiries daemon should be started first.

UNIX Platforms

UNIX releases of OrbixTalk use different library names depending on the specific implementations (Solaris, HP-UX and so on). The library name is as follows:

`libEventmt.a/sl/so`	Event Service library.

Where:

`mt`	Specifies a multi-threaded environment (Single threaded is not supported).
`sl`	Specifies a shared library for HP-UX.
`so`	Specifies a shared library for Solaris.
`a`	Specifies a shared library for AIX.

For example, applications running on HP-UX in a multi-threaded environment use the following library:

```
libEventmt.sl
```

OrbixTalk applications must be linked with the Orbix library as OrbixTalk uses Orbix functionality. The Orbix libraries have a similar naming convention to OrbixTalk libraries. For example:

```
liborbixmt.a/so/sl
```

Refer to the demos in the `demo` directory and the generalized makefile, `orbixtalk_xxx.mk`, for the exact build requirements for your specific platform.

Microsoft Windows Platforms (WIN32)

Microsoft WIN32 platforms, such as Windows NT, are primarily multi-threaded. This is the available library:

`ITOEI.LIB`	Dynamic multi-threaded Event Service console-based library.

OrbixTalk applications must be linked with the Orbix library as OrbixTalk uses Orbix functionality. The available Orbix libraries include:

`ITMI.LIB`	Dynamic multi-threaded Orbix library.
`ITMxxx.DLL`	(xxx depends on Orbix version).

To build an OrbixTalk application, you must have the following in your VC++ project settings :

1. Add `ORBIX_DLL` and `ORBIXTALK_DLL` preprocessor definitions to **General** category of the C/C++ page.
2. Set the run-time library to Multithreaded DLL in **Code Generation** category of the C/C++ page.
3. Add the `itmi.lib` and `itotmi.lib` libraries to **General** category of **Link** page.
4. Add the `itoei.lib` to **General** category of **Link** page.

13

Daemons

This chapter provides information about the OrbixTalk Directory Enquiries daemons (otd and otdsm) and the OrbixTalk MessageStore daemon (otmsd).

Overview

OrbixTalk uses a number of daemons that provide services to all OrbixTalk applications. This chapter provides information about the following daemons:

- The OrbixTalk Directory Enquiries daemon (`otd`) provides mappings between topic names and multicast addresses for all OrbixTalk applications. The OrbixTalk Directory Enquiries daemon (`otdsm`) enables you to view information about topics and applications.
- The OrbixTalk MessageStore daemon (`otmsd`) acts as an intermediary in the Store and Forward protocol.

For information about configuration parameters relating to the OrbixTalk daemons, see Appendix A, "Configuration Parameters".

Note: The command line switches are common to all the daemons, and are explained in detail for the `otd` in the following section.

Using the OrbixTalk Directory Enquiries Daemon (otd)

The otd must be present in every OrbixTalk system. Applications contact the otd to get unique application identifiers, and to map between topic names and multicast addresses. The otd must execute before any other OrbixTalk application, or the MessageStore daemon.

Usage:

```
otd -v
otd -h
otd -?

otd   [-s] [-d] [-F]
      [-i] [-u] [-D] (NT only)
      [-B] (UNIX platforms only)
```

Getting System Information

Use the -v switch to print version and configuration information to the standard output then exit. This includes the product code, build date and time, and the current settings for each of the configuration items as well as their name and a short description of each.

The -v switch is useful for checking your system configuration. IONA support generally requests the output of otd -v when you submit a query on OrbixTalk.

Getting Usage Information

Use the -h or -? switches to output information on the command line switches available for the otd.

Other Switches

-s	Starts the daemon in slave mode for fault-tolerance/fail-over support.
-d	Disables demotion of a slave mode daemon in the presence of a master mode daemon.
-F	Executes the daemon as a foreground process.
-B	Executes the daemon as a background process (UNIX platforms only).

`-i`	Installs the daemon as a service (NT only).
`-u`	Uninstalls the daemons as a service (NT only).
`-D`	Installs a dual daemon service–master and slave (NT only).

Security Considerations When Running the OrbixTalk Daemon (otd)

Unlike the Orbix Daemon (`orbixd`) where security must always be a major consideration, the `otd` is not responsible for actually starting any other processes. For this reason, security issues with the `otd` are of a different nature.

As the `otd` usually have a number of users, the main issue with security is the potential shut-down of the daemon (either accidentally or intentionally) as this has potential to affect other users who are using the same `otd`.

This is an issue that system administrators can address by giving the correct privileges to appropriate users.

Fault Tolerance Support

OrbixTalk supports fault tolerance facilities which, if enabled, allow the OrbixTalk MessageStore daemon (`otmsd`) and the OrbixTalk Directory Enquiries daemon (`otd/otdsm`) pairs to be run simultaneously.

For more information about fault tolerance, refer to Chapter 14, “Fault Tolerance”.

NT Service Support

OrbixTalk allows the `otd` to be run as a Windows NT service. The OrbixTalk daemon is installed as a service using the `-i` switch as follows:

```
otd -i
```

This command installs the OrbixTalk daemon as a service called `OrbixTalk OTD`. The OrbixTalk daemon can then be started, stopped and paused from the **Services** facility in the control panel.

Use the -D switch to install a dual OrbixTalk daemon service to support Fault Tolerance as follows:

```
otd -D
```

This command installs a master mode OrbixTalk daemon called OrbixTalk OTD Master, and a slave mode OrbixTalk daemon called OrbixTalk OTD Slave.

The Orbix.cfg entry or environment variable IT_OT_DAEMON_BACKGROUND must be set to 1 before starting the OrbixTalk daemon from the **Services** menu in the control panel. Similarly, IT_LOG_SYSLOG should be set to 1 and IT_LOG_CONSOLE set to 0 to redirect output to a log file contained in the directory specified by IT_APP_STORE. For more information about these configuration parameters, refer to Appendix A, "Configuration Parameters".

Once the OrbixTalk daemon is installed, modify the start-up to interact with the desktop as follows:

1. Double-click on each OrbixTalk daemon entry displayed to view the Allow **Service** to Interact with **Desktop** toggle button.
2. Click on the **Allow Service** to Interact with **Desktop** toggle button to start or stop the interaction.

Use the -u switch to uninstall any of the OrbixTalk daemons as follows:

```
otd -u
```

This command uninstalls any of the following OrbixTalk daemons:

```
OrbixTalk OTD
OrbixTalk OTD Slave
OrbixTalk OTD Master
```

Daemon Support for UNIX

On UNIX platforms, the otd can be run as a background process that is completely disassociated from the controlling terminal. Set the IT_OT_DAEMON_BACKGROUND configuration parameter to 0 or 1 to determine the execution of the OrbixTalk daemon as follows:

0 (default)	otd/otdsm runs in the foreground.
1	otd/otdsm runs as a background process that has no association with the terminal that executed the process.

Use the -F command line switch to override the IT_OT_DAEMON_BACKGROUND configuration parameter. This causes the otd to run as a foreground process.

Use the -B switch to force an OrbixTalk daemon to run in the background.

Using the OrbixTalk Directory Enquiries Daemon (otdsm)

Use the otdsm in place of the standard otd to observe the state of the system.

Usage:

```
otdsm
```

The otdsm supports all the command line switches used by the otd.

If fault tolerance is enabled for an otdsm, the -d switch is the enforced default. This switch disables demotion of the otdsm once it has become primary. The master-slave relationship is affected such that the master mode daemon will not automatically become primary in the presence of a slave mode daemon. Instead, the slave mode daemon will remain primary and the master mode daemon will hold off becoming primary until the Slave exits or fails.

Using the OrbixTalk MessageStore Daemon (otmsd)

The otmsd is required in systems that use the OrbixTalk Store and Forward protocol (otsfp). The otmsd acts as an intermediary process for communication, storing messages persistently to guarantee message delivery.

Usage:

```
otmsd
```

The otmsd supports all the command line switches used by the otd, including those related to fault tolerance, NT service and daemon support for UNIX.

For more information about the OrbixTalk MessageStore daemon (otmsd), refer to Chapter 3 “OrbixTalk MessageStore”.

14

Fault Tolerance

This chapter describes how to implement fault tolerance for OrbixTalk daemons and OrbixTalk applications.

Overview

OrbixTalk supports fault tolerance facilities which, if enabled, allows a pair of OrbixTalk MessageStore daemons (`otmsd`) and a pair of OrbixTalk Directory Enquiries daemons (`otd/otdsm`) to be run simultaneously. In the event of one daemon in either pair failing, the remaining daemon takes over. OrbixTalk-based applications also use inherent fault tolerant mechanisms, such as retry attempts to contact the daemons, to build an overall fault tolerant system.

Note: In this chapter, OrbixTalk daemon refers to the OrbixTalk Directory Enquiries daemon (`otd`), the Browsable Directory Enquiries daemon (`otdsm`) and the OrbixTalk MessageStore daemon (`otmsd`.).

Figure 14.1 shows a recommended configuration for OrbixTalk with fault tolerance support enabled:

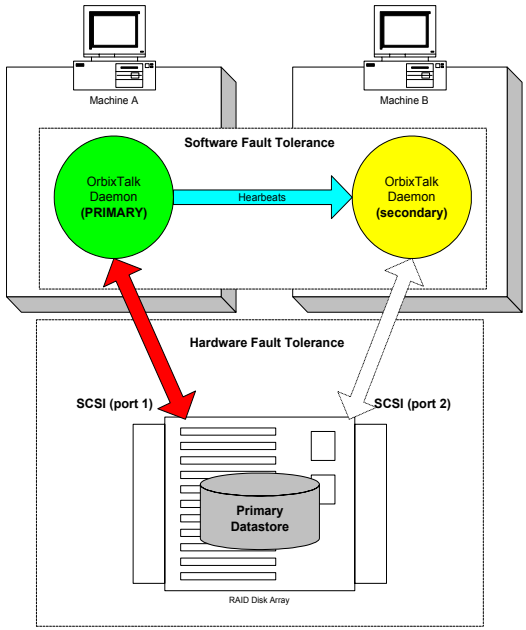

Figure 14.1: *OrbixTalk Configuration with Fault Tolerance Supported*

A Redundant Array of Inexpensive Disks (RAID) provides a datastore that enables a disk in the RAID to be removed without incurring loss of data. A RAID in the system provides more reliability and ease of recovery. The configuration can be scaled from a dual SCSI port RAID down to a hard disk residing in a server. The only requirement for the datastore(s) is that each OrbixTalk daemon must have access. If performance is an issue, each OrbixTalk daemon must be able to access the datastore efficiently.

Both OrbixTalk daemons, comprising a fault tolerant pair, share the same datastore(s). OrbixTalk ensures that only the OrbixTalk daemon in primary phase is allowed to access the datastore at any given time.

Figure 14.1 shows Machine A running the master mode OrbixTalk daemon in primary phase and Machine B running the slave mode OrbixTalk daemon in secondary phase. Each daemon is either in master or slave mode. Only one of each type (master or slave) is able to run on the same base multicast IP address and port number.

Machine A, running the master mode OrbixTalk daemon in primary phase, emits a periodic heartbeat ping (multicast network packet), to indicate it is healthy. OrbixTalk daemons ping at regular, configurable, intervals to allow their fault tolerant partner to detect them and know they are responding normally. Heartbeat pings contain information about whether the OrbixTalk daemon is in primary phase or secondary phase. Secondary phase heartbeat pings are emitted when an OrbixTalk daemon is in transitory phase. The Daemon Process Detection tool (`otpsd`) reports this information.

Machine B running the slave mode OrbixTalk daemon in secondary phase monitors the heartbeat ping. If a number of heartbeat pings are missed, the OrbixTalk daemon on machine B changes to primary phase. The number of heartbeat pings is configurable. In this event, the master mode OrbixTalk daemon has either exited, hung up or is in some other failed state (software failure).

If the slave mode OrbixTalk daemon becomes primary then later detects that the master mode OrbixTalk daemon is present once more, the slave mode OrbixTalk daemon returns to secondary phase and enables the master mode OrbixTalk daemon to become primary. In this way, a slave mode OrbixTalk daemon remains a slave for backup/fail-over.

By default, an OrbixTalk daemon starts in master mode. To start the OrbixTalk daemon in slave mode, use the `-s` switch.

Use the `-d` switch to ensure that a slave mode OrbixTalk daemon entering primary phase cannot return to secondary phase in the presence of the master mode OrbixTalk daemon. For example:

- Machine A is running the master mode OrbixTalk daemon. It has the necessary resources (CPU, memory, and so on) to support directory enquiries for the entire system. Machine B is running the slave mode OrbixTalk daemon providing support in case the master mode OrbixTalk daemon fails.

- Both machines use a common RAID array to store messages for the daemon. Both daemons are run from scripts that restart the processes if they fail.
- If Machine A fails, the slave mode OrbixTalk daemon on Machine B is promoted to primary phase and takes over the processing as the master mode OrbixTalk daemon.

  When Machine A starts processing again, the daemon is restarted. One of the following occurs:

  - The `-d` switch is not specified.

    The OrbixTalk daemon on Machine B detects that the OrbixTalk daemon on Machine A has restarted and demotes itself to secondary phase. This is useful when Machine B does not fully support the processing required by the system or machine A is more suited to the task.
  - The `-d` switch is specified.

    If Machine B has the same resources as Machine A, then the system is not concerned which machine runs the master mode Orbixtalk daemon. In this case, the `-d` switch prevents the slave mode OrbixTalk daemon on Machine B from demoting itself when the OrbixTalk daemon on Machine A restarts. The OrbixTalk daemon that restarts on Machine A remains as a slave mode OrbixTalk daemon while the daemon on machine B serves the system. This means less delays in the system as no negotiation is required to determine which OrbixTalk daemon should be in primary phase.

Transition Diagrams

Figure 14.2 shows the transition phases for the master mode and slave mode OrbixTalk daemons:

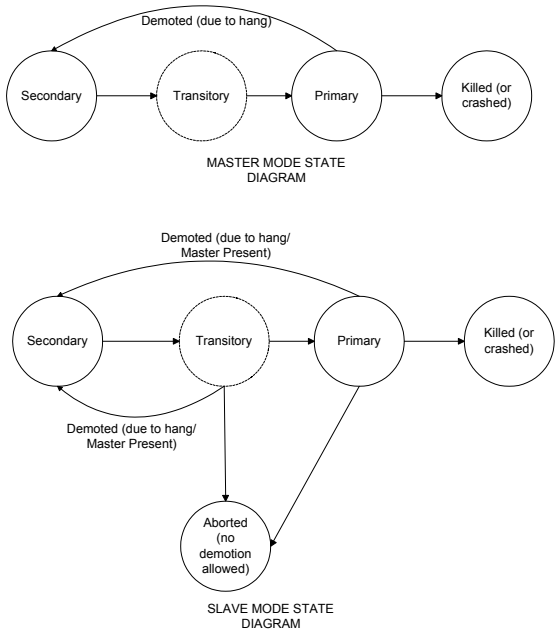

Figure 14.2: *Transition Phases for Master and Slave Mode Daemons*

Summary of Phases

Secondary Phase

- Unresponsive to OrbixTalk talkers/listeners.
- Normally does not emit heartbeat pings unless required on response from `otpsd` or by another OrbixTalk daemon initializing.

Transitory (pre-primary) Phase

- Unresponsive to OrbixTalk talkers/listeners.
- Emits secondary phase heartbeat pings.
- Performs lock checks to detect another transitory or primary phase daemon before going to primary phase.

Primary Phase

- Responsive to OrbixTalk talkers/listeners.
- Constantly emits primary phase heartbeat pings so another OrbixTalk daemon can detect a failure.
- Performs lock checks to detect another transitory or primary phase daemon.

Types of Failure

Fault tolerance in OrbixTalk addresses three problem areas:

- The OrbixTalk daemon process dies due to a software failure.
- The host machine running the OrbixTalk daemon dies due to a hardware failure.
- The host machine running the OrbixTalk daemon cannot be reached due to a network failure.

Software Failure

The slave mode OrbixTalk daemon monitors the activity of the master OrbixTalk daemon. When a software failure occurs in one of the OrbixTalk daemons and the master mode OrbixTalk daemon dies, the slave mode OrbixTalk daemon is promoted to primary phase.

A mechanism must be in place that re-launches the master mode OrbixTalk daemon on Machine A or the slave mode OrbixTalk daemon running on Machine B if it has failed. For example, a simple looping script (Windows NT/ UNIX) or adding an `inittab` entry with respawn as an action field (UNIX only).

The OrbixTalk `otpsd` can be used to detect an OrbixTalk daemon that is hanging in primary or secondary phase; that is, a daemon that is no longer heart-beating/pinging. This enables the existing process to be killed and restarted. OrbixTalk daemons respond to the `otpsd` in primary and secondary phase provided the `-e` switch is supplied to the `otpsd` tool to make the secondary daemon visible. A shell script (or Windows NT batch file) can be written to kill an OrbixTalk daemon that is hanging for long periods of time, but should enable OrbixTalk daemons to resolve the hang-up themselves by demotion or exit.

Hang Detection

An environment that cannot guarantee responsiveness of OrbixTalk daemons (whether it be `otd`, `otdsm` or `otmsd`) at all times can lead to exits or demotions if the hang-ups are of sufficient length to allow a daemon to be half way toward changing phase from secondary to primary. An OrbixTalk daemon that is hanging occasionally prints a warning as follows:

```
...
May 27 16:07:09 otd: Warning: daemon was unresponsive for
approximately 3585 ms.
...
```

These warnings occur if the system is heavily loaded or the `IT_FT_HEART_BEAT_INTERVAL` and `IT_MAX_FT_HEART_BEAT` configuration parameters are set incorrectly. The OrbixTalk daemon in secondary phase can enter primary phase or an OrbixTalk daemon can detect hanging and enter transitory phase where lock checking begins. The OrbixTalk daemon in primary phase is detected and the OrbixTalk daemon in transitory phase is demoted or exits (depending on the presence of the `-d` switch).

For an OrbixTalk daemon to enter transitory phase due to another OrbixTalk daemon hanging, the duration of the hang needs to be greater than:

$$\frac{\texttt{IT\_FT\_HEART\_BEAT\_INTERVAL} \times (\texttt{IT\_MAX\_FT\_HEART\_BEAT} - 1)\ \texttt{ms}}{2}$$

To reduce the possibility of hanging, set the `IT_FT_HEART_BEAT_INTERVAL` and `IT_MAX_FT_HEART_BEAT` configuration parameters in the `Orbix.cfg` file appropriately. For more information about the `IT_FT_HEART_BEAT_INTERVAL` and `IT_MAX_FT_HEART_BEAT` configuration parameters, refer to Appendix A, "Configuration Parameters". Alternatively, a server with faster CPU(s), more memory or in a generally less loaded state can be required.

Hardware Failure

OrbixTalk supports recovery from hardware failure provided the master and slave OrbixTalk daemons are running on separate hosts. When an OrbixTalk daemon, running on host A, fails then the other OrbixTalk daemon, running on host B, takes the incoming requests/messages.

Using a RAID unit to maintain a single datastore for each OrbixTalk daemon pair ensures that the system survives in the event of a single disk failure.

Network or SCSI Cable/Port Failure

In the event of a failure between the RAID and either host (SCSI cable or port failure), the affected OrbixTalk daemon is unable to access the datastore. It is possible to configure certain types of RAIDs to use dual SCSI ports so that a separate data path connects each host, running an OrbixTalk daemon, to the RAID.

Situations where network failures arise are shown in Figure 14.3 and Figure 14.4.

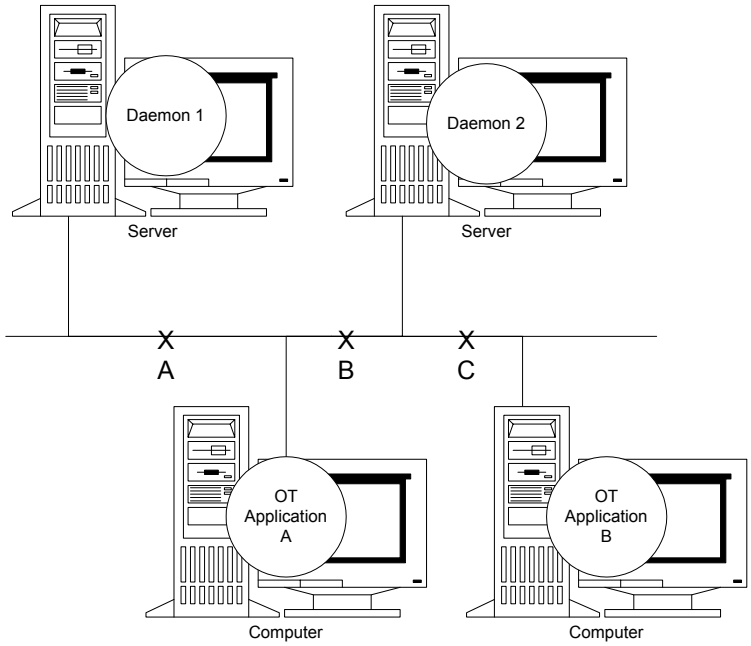

Figure 14.3: *Network Failures*

A physical break at C has no effect on the OrbixTalk daemon. However, OT Application B cannot communicate with either Daemon 1 or Daemon 2. The OT Application B fails.

If there is a break at A or B, OT Application A and OT Application B still access either Daemon 1 or Daemon 2.

When a break at A occurs, the OrbixTalk daemon currently in the secondary phase becomes primary because it no longer receives heartbeat pings from the other OrbixTalk daemon; both OrbixTalk daemons are now primary. This is potentially dangerous and leads to both OrbixTalk daemons aborting.

A break at B can lead to more serious problems. OT Application A continues using Daemon 1 and OT Application B uses Daemon 2. A shared datastore is used. The datastore can become inconsistent as applications bind more topics, new applications on either side of the breakage are started or messages are sent to a MessageStore from either side of the breakage. To avoid these situations, place the OrbixTalk daemons at one end of the network with no OrbixTalk applications running on those servers as shown in Figure 14.4:

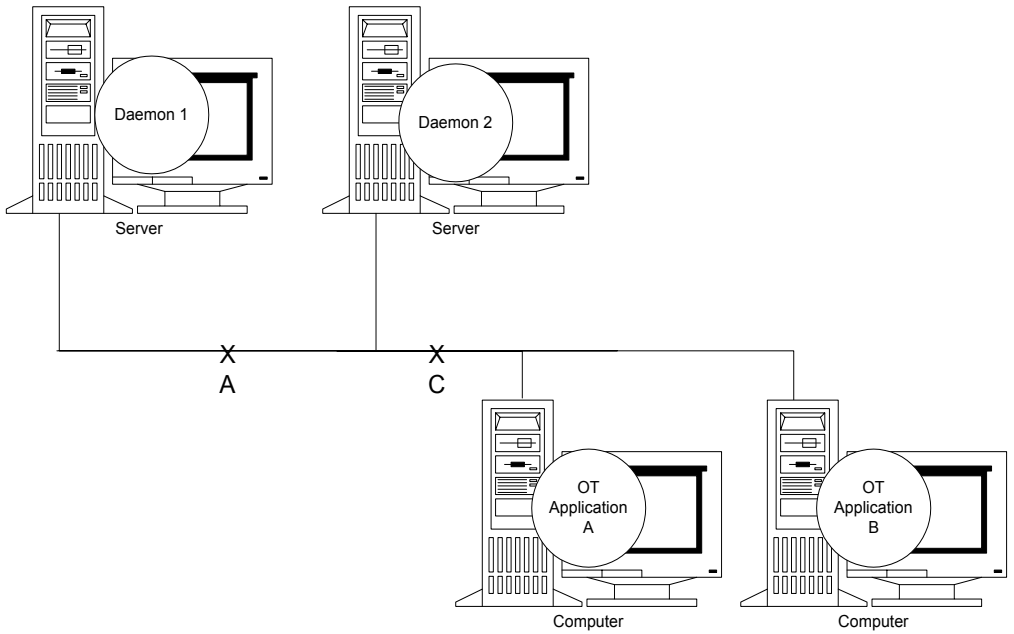

Figure 14.4: *Recommended Configuration for OrbixTalk Daemons*

This is the safest solution and means total failure of applications rather than partial failure. OrbixTalk does not provide for recovery from all types of network failure but does safeguard against corruption of shared datastores. Placing the OrbixTalk daemons at the end of the network is recommended.

Lock Checking

In the event of a network failure, as shown in Figure 14.3 and Figure 14.4, OrbixTalk prevents the OrbixTalk daemon in secondary phase changing to primary phase. When an OrbixTalk daemon is in transitory or primary phase, the datastore uses a locking mechanism in the form of a dynamically changing lock file to enable an OrbixTalk daemon to recognize that an OrbixTalk daemon in primary phase is already active.

Due to network delays or excessive loading, an OrbixTalk daemon can begin to transition into primary phase when another OrbixTalk daemon is already in primary phase. The OrbixTalk daemon in transitory phase is demoted or exits to avoid a critical fault being registered in the lock file (causing both OrbixTalk daemons to exit).

When two OrbixTalk daemons are in primary phase, a critical fault is registered within the lock file and both OrbixTalk daemon(s) exit. This ensures that datastores are guarded from network errors causing serious corruption but manual intervention to correct the problem is required. The lock file must be removed to allow each OrbixTalk daemon to restart once the network has been repaired (as a safety precaution). For information about the configuration parameters used by lock files, refer to Appendix A, “Configuration Parameters”.

Lock files exist after the OrbixTalk daemons have exited. Lock files only need to be removed when a network failure has occurred and the OrbixTalk daemon pair(s) cannot start.

15

OrbixTalk System Exceptions

The OrbixTalk API generates exceptions when error or fault conditions arise. This chapter defines the range of exceptions that can be raised by OrbixTalk, and describes the conditions under which they are raised.

A basic understanding of OrbixTalk concepts and OrbixTalk programming is assumed.

Overview

OrbixTalk uses the exception mechanism of C++ to indicate abnormal conditions. Fault or error conditions can arise because of a poorly configured system, a network fault, file system fault, or internal OrbixTalk error. The type of exceptions that can be produced by OrbixTalk need to be known in advance so that an OrbixTalk application can be designed to cope with them.

For example, the `OrbixTalk::initialise()` function can raise a `COMM_FAILURE` exception with one of two description messages:

```
Could not create OrbixTalk object
Could not create channel table
```

In general, COMM_FAILURE exceptions indicate a fatal error that cannot be recovered from within the OrbixTalk system. However, it is likely that exceptions found in a system during testing can be corrected through changes to the configuration parameters used for OrbixTalk, or by modifying the way in which the system is programmed.

OrbixTalk API Exceptions

This section describes the exceptions that are generated from specific functions in the OrbixTalk Application Programming Interface (API) and includes a short description of the reason for the exception.

OrbixTalk::addTimerEvent()

```
OrbixTalk::addTimerEvent
(
  TimerEvent*         pTimer,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)

Description "OrbixTalk is not initialised"

This exception is generated when OrbixTalk::initialise() has not been called, or when OrbixTalk::terminate() has already been called.

Description "Could not add timer"

This exception is generated when the timer could not be added to the timer service of OrbixTalk. This indicates an internal error.

Type BAD_PARAM, NO_OBJ_NAME

This exception is generated when the TimerEvent* pTimer parameter does not point to a valid timer.

OrbixTalk::initialise

```
OrbixTalk::initialise
(
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)

Description "Could not create OrbixTalk object"

This exception is generated when OrbixTalk fails to initialise. This is the result of an internal error.

Description "Could not create channel table"

This indicates an internal OrbixTalk error.

OrbixTalk::isIdle()

```
OrbixTalk::isIdle
(
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)

Description "OrbixTalk is not initialised"

This exception is generated when `OrbixTalk::initialise()` has not been called, or when `OrbixTalk::terminate()` has already been called.

OrbixTalk::isRegistered()

```
OrbixTalk::isRegistered
(
  CORBA(Object_ptr)   pObject,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)

Description "OrbixTalk is not initialised"

This exception is generated when `OrbixTalk::initialise()` has not been called, or when `OrbixTalk::terminate()` has already been called.

Type BAD_PARAM, NO_OBJ_NAME

This exception is generated when the `CORBA::Object_ptr` parameter does not specify a valid proxy object.

OrbixTalk::registerListener()

```
OrbixTalk::registerListener
(
  CORBA(Object_ptr)   pObject,
  REPLAY_TYPE         replayStore = REPLAY_ALL,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type `COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)`

Description "OrbixTalk is not initialised"

This exception is generated when `OrbixTalk::initialise()` has not been called, or when `OrbixTalk::terminate()` has already been called.

Description "Object is already registered"

This exception is generated when the `CORBA::Object_ptr` parameter is already registered as a talker or listener.

Type `BAD_PARAM, NO_OBJ_NAME`

This exception is generated when the Topic Name associated with the listener object is empty.

Type `INV_OBJREF, IS_A_PROXY`

This exception is generated when the `CORBA::Object_ptr` parameter does not specify a valid proxy object.

OrbixTalk::registerTalker()

```
OrbixTalk::registerTalker
(
  CORBA(Object_ptr)   pObject,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type `COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)`

Description "OrbixTalk is not initialised"

This exception is generated when `OrbixTalk::initialise()` has not been called, or when `OrbixTalk::terminate()` has already been called.

Description "Object is already registered"

This exception is generated when the CORBA::Object_ptr parameter is already registered as a talker or listener.

OrbixTalk::registerTalker()

```
OrbixTalk::registerTalker
(
  const char*         pServerName,
  const char*         pTypeName,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)

Description "OrbixTalk is not initialised"

This exception is generated when OrbixTalk::initialise() has not been called, or when OrbixTalk::terminate() has already been called.

Description "Object is already registered"

This exception is generated when the CORBA::Object_ptr parameter has already been registered as a talker or listener.

Type BAD_PARAM, NO_OBJ_NAME

This exception is generated when the marker for the proxy object is not in the form of a valid Topic Name.

Type INV_OBJREF, IS_A_PROXY

This exception is generated when the CORBA::Object_ptr parameter does not specify a valid proxy object.

OrbixTalk::removeTimerEvent()

```
OrbixTalk::removeTimerEvent
(
  TimerEvent*         pTimer,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)

Description "OrbixTalk is not initialised"

This exception is generated when `OrbixTalk::initialise()` has not been called, or when `OrbixTalk::terminate()` has already been called.

Description "Could not remove timer"

This exception is generated when the timer could not be removed from the timer service of OrbixTalk. This indicates an internal error.

Type `BAD_PARAM, NO_OBJ_NAME`

This exception is generated when the `TimerEvent* pTimer` parameter does not point to a valid timer.

OrbixTalk::setPersistentAppName()

```
OrbixTalk::setPersistentAppName
(
  const char*         pName,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type `COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)`

Description "OrbixTalk is not initialised"

This exception is generated when `OrbixTalk::initialise()` has not been called, or when `OrbixTalk::terminate()` has already been called.

Description "Failed to set application name"

This indicates that the persistent application name could not be set. It could be a result of an invalid application name being used, or the inability to create the necessary state files.

OrbixTalk::unregister()

```
OrbixTalk::unregister
(
  CORBA(Object_ptr)   pObject,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
)
```

Type `COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)`

Description "OrbixTalk is not initialised"

This exception is generated when `OrbixTalk::initialise()` has not been called, or when `OrbixTalk::terminate()` has already been called.

Description "Object is not registered"

This exception is generated when the object that is being unregistered is not registered as a talker or listener.

Type `BAD_PARAM, NO_OBJ_NAME`

This indicates an internal Orbix/OrbixTalk error.

General Exceptions

The following exceptions can be raised using the environment passed to `CORBA::Orbix.processEvents()` or `CORBA::impl_is_ready()`. Each exception is not related to a specific OrbixTalk API function.

Type `COMM_FAILURE, PUBLISH_SUBSCRIBE (10098)`

Description "Operation is not oneway"

This exception is generated when you attempt to invoke a method on an OrbixTalk proxy that is not oneway. For example, it is raised if you attempt to perform a two-way method invocation on the OrbixTalk proxy.

Description "Object not registered as talker"

This exception is generated when you attempt to invoke a method on an object that is not registered as a talker.

Description "1 - message too large"

This exception is generated when the contents of a message is greater than the limit set by the `IT_MAX_MSG_SIZE_KB` configuration parameter.

Description "3 - no such field"

This exception is generated by the `otadmin` tool on an internal error related to communication with the OrbixTalk Directory Enquiries daemon. This can indicate a mismatch between the versions of `otadmin` and `otd/otdsm`.

Description "4 - end of file prematurely reached"

This exception is generated by the OrbixTalk MessageStore daemon (`otmsd`) when disk error occurs on compaction or on adding records to the MessageStore.

Description "10 - bad name"

This exception is generated by the OrbixTalk tools when parsing command line arguments that are incorrect. On joining a multicast group with an invalid client, this indicates an internal error.

Description "12 - could not create socket"

This exception is generated when the system fails to create a multicast socket. This indicates an internal error, or can indicate that the process has run out of file descriptors. On UNIX platforms, the `ulimit` command can be used to increase the number of file descriptors available to a process.

Description "13 - setsockopt failure for `SO_RCVBUF`"

This indicates an internal error generated when the receive buffer could not be created for a socket.

Description "14 - setsockopt failure for `SO_SNDBUF`"

This indicates an internal error generated when the send buffer associated with a socket could not be created.

Description "15 - setsockopt failure for `SO_REUSEADDR`"

This exception is generated when the system fails to set reuse for a socket. This indicates an internal error.

Description "16 - could not bind socket"

This exception is generated when the system fails to bind a socket. This indicates an internal error.

Description "17 - name bind failure"

This exception is generated when the system fails to bind a Topic Name to the topic. This indicates an internal error.

Description "18 - setsockopt failure for `IP_ADD_MEMBERSHIP`"

This exception is generated when a `setsockopt()` call fails internally.

Description "19 - setsockopt failure for `IP_DROP_MEMBERSHIP`"

This exception is generated when a `setsockopt()` call fails internally.

Description "20 - sendto failed"

"21 - rcvfrom failed"

These exceptions are generated when network communication fails on a socket. This indicates either a network interface failure, or an internal error.

Description "23 - gethostname/gethostent failed"

This exception is generated when the `gethostname()` or `gethostent()` system call fails. This could indicate a badly configured operating system.

Description "25 - no such part"

This indicates an internal error for the OrbixTalk Directory Enquiries daemon. It indicates a problem with the internal memory structures.

Description "29 - data lost"

This exception is generated by a listener using the Reliable Multicast Protocol (`otrmp`) when the listener determines that data has been lost. To correct this problem, change the configuration parameters related to the reliability of the Reliable Multicast Protocol described in Appendix A, "Configuration Parameters".

Description "30 - directory server is uncontactable"

This exception is generated when an application cannot contact the OrbixTalk Directory Enquiries daemon (`otd`). This can be because the `otd` is not running, or the configuration parameters are not consistent in the system, or there is a problem with network configuration.

Description "31 - opening log"

This exception is generated when the system fails to open a log file. This exception can arise when the parameter specifying the directory in which the log file should be used is incorrect, or points to a directory that does not have the correct access permissions.

Description "32 - log entry already exists"

This indicates an internal error that can be produced by the OrbixTalk Directory Enquiries daemon.

Description "34 - log seek failed"

This indicates an internal error relating to log files.

Description "35 - log read failed"

This indicates an internal error relating to log files.

Description "36 - log write failed"

This exception is generated by the OrbixTalk MessageStore daemon when an internal error occurs that relates to the databases used by the OrbixTalk MessageStore daemon.

Description "37 - log entry not found"

This exception is generated when the OrbixTalk MessageStore daemon fails to find information about a particular application or topic in its databases; that is, the information has been removed while a topic remains active.

Description "40 - not MessageStore group name"

This exception is generated by a talker using the Store and Forward Protocol (`otsfp`) specifying a bad MessageStore Topic Name. This can be corrected by modifying the configuration parameter `IT_MS_TOPIC_NAME`.

Description "44 - An OrbixTalk message using the Store and Forward Protocol has been rejected. (The OrbixTalk MessageStore daemon (`otmsd`) process can have failed)."

This exception is generated when the OrbixTalk MessageStore daemon (`otmsd`) cannot be contacted by a talker application when it attempts to send a message. This can be because the `otmsd` is not running, or the applications are using inconsistent configuration parameters, or there is a network problem.

Description "45 - Invalid argument supplied"

This exception is generated by the `otadmin` tool when an invalid argument is used, or by the OrbixTalk MessageStore daemon when an internal error occurs.

Description "46 - set time to live failed"

This exception is generated when the system call that modifies the time to live for UDP packets fails.

Description "47 - internal error - log index not found"

This indicates an internal error for the OrbixTalk Directory Enquiries daemon.

Description "49 - corrupt log frame"

This exception is generated when state log file entries have become corrupted.

Description "50 - invalid field type"

This indicates an internal error that can be produced by the OrbixTalk Directory Enquiries daemon and/or the OrbixTalk MessageStore daemon when decoding messages.

Description "51 - unexpected field type"

This indicates an internal error that can be produced by the OrbixTalk Directory Enquiries daemon and/or the OrbixTalk MessageStore daemon when decoding messages.

Description "52 - unsupported operation"

This is either an internal error on a topic, or an attempt to specify a replay type for a topic using the Reliable Multicast Protocol.

Description "53 - topic not bound"

This exception is generated when you attempt to send a message on a topic that has not been bound to a multicast address. This is either an internal error, or a problem with the application contacting the OrbixTalk Directory Enquiries daemon (`otd`). It can be corrected by ensuring that the OrbixTalk Directory Enquiries daemon can be contacted by the application.

Description "54 - message store not known"

This exception is generated when an application fails to contact the OrbixTalk MessageStore daemon. This can be the result of inconsistent configuration parameters, or a network failure.

Description "55 - attempt to re-use existing dir. enq. handler"

This indicates an internal error.

Description "58 - deletion failed"

This exception is generated by the OrbixTalk MessageStore daemon when it fails to delete an entry from the MessageStore. This indicates an internal failure.

Description "59 - initialisation failed"

This exception is generated by the OrbixTalk MessageStore daemon when it fails to initialise the MessageStore database. This could be a result of bad configuration parameters that specify the directories and names for the MessageStore files. It can be corrected by ensuring that these settings indicate directories that exist and have the correct access permissions, and that the filenames are correct.

Description "60 - backup failed"

This exception is generated by the OrbixTalk MessageStore daemon when a backup for compaction fails. This can indicate that the configuration parameter specifying the directory used for backups (`IT_MS_COMPACT_BACKUP_DIR`) is not set correctly.

Description "61 - message timed out"

This exception is generated by a talker application using the Store and Forward Protocol (`otsfp`) failing to contact the OrbixTalk MessageStore daemon (`otmsd`). See exception 54.

Description "62 - SFP data lost"

This exception is generated when the Store and Forward Protocol (`otsfp`) loses data. This error indicates a problem with the `otsfp`.

Description "63 - Out of memory"

This exception is generated when an application exhausts all available memory.

Description "64 - Internal error - Cannot wrap message"

This indicates an internal error relating to message construction.

Description "65 - Reached end of message"

This indicates an internal error relating to message receipt.

Description "66 - Attempt to listen on talk topic"

This indicates an internal error.

Description "67 - Attempt to talk on listen topic"

This indicates an internal error.

Description "68 - Failed to create thread"

This indicates an internal error.

Description "69 - Sub-system is already running"

This indicates an internal error.

Description "70 - System is not initialised"

This indicates an internal error.

Description "71 - Attempt to attach second 'phone to topic"

This indicates an internal error.

Description "72 - Invalid reference passed"

This indicates an internal error relating to messages.

Description "73 - Pending call"

This indicates an internal error generated for topics using the Store and Forward protocol (`otsfp`).

Description "77 - Duplicate message"

This indicates an internal protocol error.

Description "78 - Message out-of-sequence"

This indicates an internal protocol error.

Description "79 - Internal error"

This indicates an internal error for the OrbixTalk MessageStore daemon.

Description "80 - Failed backup"

This exception is generated by the OrbixTalk MessageStore daemon when it fails to backup the MessageStore prior to compaction.

Description "81 - General error"

This indicates an internal error for the OrbixTalk MessageStore daemon.

Description "85 - Prod received"

This indicates an internal protocol error.

16

Tools

This chapter provides information about the otdat, otadmin, and otpsd tools.

Overview

This chapter provides information about the following tools:

- The `otdat` tool enables you to view the consumer state logs and the MessageStore logs.
- The `otadmin` tool enables you to compact the MessageStore.
- The `otpsd` tool enables you to check which daemons are running.

Using the State Log Analysis Tool (otdat)

The `otdat` tool enables you to analyze the contents of the state log files (`.dat`) produced by consumers and information stored by the MessageStore daemon. The `otdat` tool enables you to dump the contents of `.dat` files and the MessageStore to `stdout` in an ASCII format. You can specify precise filters for the MessageStore information.

Usage:

```
otdat -v
otdat -h
```

```
otdat <filenames>

otdat -S|[-s msname] [-p mspath] [-c columnwidth]
[-l pagelength] [-a appnames] [-t topics]
[-e appseqnum] [-f topicseqnum] [-o TTSN|TAASN|TIME]
[-d dd/mm/yy:hh:mm-dd/mm/yy:hh:mm]
[-r columnOrder]
```

Note: The OrbixTalk Directory Enquiries daemon (`otd/otdsm`) must be running to use this tool.

Dumping to the Standard Output (stdout)

To dump the contents of one or more consumer state logs, enter this command:

```
otdat <filenames>
```

where `<filenames>` are the `.dat` files to be displayed. Only data files with OrbixTalk standard names are allowed, for example:

```
otdat OT-talk5-Talk.dat OT-talk8-Talk.dat
```

The syntax for dumping the contents of the MessageStore is as follows:

```
otdat -S|[-s msname] [-p mspath] [-c columnwidth]
[-l pagelength] [-a appnames] [-t topics]
[-e appseqnum] [-f topicseqnum] [-o TTSN|TAASN|TIME]
[-d dd/mm/yy:hh:mm-dd/mm/yy:hh:mm]
```

Where:

`-S`	Specifies the MessageStore dump option and prompts `otdat` to use the default MessageStore specified in the configuration file.
`-s`	Specifies the MessageStore dump option and the name of the MessageStore.
`-p`	Specifies the path to the MessageStore (used with the `-s` option).
`-c`	Number of columns for output of application and topic names. Default is 30.
`-l`	Number of lines per page. A title is inserted at the top of each page. If this value is not specified then the output is continuous.

`-a` Specifies ranges of application names or ID's to view. Each range must be separated by a comma and have no white space. For example:

```
-a //OT/supplier5,//OT/supplier9,4EF-4FF
```

`-t` Specifies ranges of topic names or ID's to view. Each range must be separated by a comma and have no white space. For example:

```
-t otsfp//OT/music/rock,CF4,otsfp//OT/music/jazz
```

`-e` Specifies ranges of application sequence numbers to view. Each range must be separated by a comma and have no white space. For example:

```
-e 1-FF,D56-DFF
```

`-f` Specifies ranges of topic sequence numbers to view. Each range must be separated by a comma and have no white space. For example:

```
-f F00-FFF,1-FF
```

`-o` Specifies an ordering for the output from the above filters:

`TTSN` - TopicID-TopicSequenceNumber.

`TAASN` -TopicID- ApplicationID--Application SequenceNumber.

`TIME` -Time-TopicID (Default).

`-d` Specifies time ranges to dump. The ranges must be complete and have no white space. For example:

```
-d 28/01/97:07:30-29/01/97:08:30,28/02/97:07:30-29/
02/97:08:30
```

`-r`	Specifies the arrangement of the columns. For example: `-r D:AI:T` Where: A: Application AI: Application ID AS: Application Sequence Number D: Datestamp T: Topic TI: Topic ID TS: Topic Sequence Number

All numeric ranges (except time) are specified in hexidecimal.

Using the MessageStore File Compaction Tool (otadmin)

This utility allows the user to compact and/or back up the Message Store database. Compaction is achieved by copying all messages greater than the specified date and time to a copy database and on completion making that copy database live. The user can also specify the topic on which to talk to the Message Store.

Usage:

```
otadmin -B|-d dd-mm-yyyy -t hh:mm|-o seconds
[-T MessageStoreTopic] [-y] [-N] [-r seconds][-i compact by topic]
|[-v]|[-h|-?]
```

Where:

`-B`	Only back up the Message Store (no compaction is performed).
`-C`	Compact the Message Store (default).
`-N`	Do not back up the Message Store for compaction. By default the message store is backed up.

`-T`	Specifies the topic on which to talk to the Message Store. If the user does not specify a topic, a default will be obtained from the system configuration information.
`-d`	Specifies a date where all messages prior to the date are removed from the Message Store. This is used in conjunction with the `-t` option to pinpoint which messages to delete. The date is in the format dd-mm-yyyy, where dd is the numerical day of the month, mm is the numerical month of the year and yyyy is the year (for example, 01-10-1997).
`-t`	In conjunction with the date, this specifies the exact time where messages prior to the time for the given date are deleted. The time is in the form hh:mm, where hh is the hour of the day (in 24 hour time) and mm is the minutes past the hour (for example, 01:35, 23:12).
`-o`	Alternative to using `-d` and `-t`. Remove all messages which are older than the number of seconds specified. This would normally be used for testing purposes.
`-y`	Specifies that user confirmation of the Compaction is not required. This is useful for `cron` entries.
`-r`	Reschedule this request for the number of seconds specified. This is only recommended for testing purposes (as all subsequent requests will be ignored).
`-i`	Specify the topics that are compactable. If this is not specified all topics will be compactable. For example: `-i otsfp//IONA,otsfp//IBM`
`-v`	Outputs product code and version information and exits.
`-h`	Outputs user Help information.
`-?`	Outputs user Help information.

Using the Daemon Process Detection Tool (otpsd)

This utility detects any running daemons (`otd` and `otmsd`) which are using the specified Directory Enquiries IP address and port. These details are taken from the `Orbix.cfg` file by default (`IT_DIRENQ_IPADDR` and `IT_DEFAULT_DIRENS_PORT` respectively) but can be overridden if supplied on the command line. The default time spent monitoring for daemons before a report is generated is 5 seconds. The `-t` switch can be used to change this time-out value.

Usage:

```
otpsd [-t nnn] [-a ipaddress] [-p portnumber]
```

Where:.

`-e`	The default behavior for `otpsd` is to display the current Primary phase OrbixTalk Daemon along with the host `ip` address, hostname and OrbixTalk Daemon description (`otd` or `otmsd`). This switch changes `otpsd's` behaviour to display the active primary OrbixTalk Daemon and any secondary phase OrbixTalk Daemon (for example, a Slave Daemon currently in secondary phase). Extra information, including the version, mode, phase and process identifier, is also displayed for each detected OrbixTalk Daemon.
`-t`	Specifies a timeout, in seconds (nnn), during which `otpsd` waits for heartbeats from the `otd` and `otmsd` daemons.
`-a`	Overrides the directory enquiries IP address specified in the `Orbix.cfg` file. This IP address is used for heartbeating (the means of detecting other daemons of the same kind for fault tolerance).
`-p`	Specifies a port (nnn) to listen on for heartbeats (see `-a` option).

17

Troubleshooting

This chapter provides answers to frequently asked questions.

Question: Orbix Compatibility

Which version(s) of OrbixTalk is compatible with each version of Orbix?

Answer Please refer to the OrbixTalk 3.0 Release Notes for full details on this subject.

Question: Listeners on Different Subnets

OrbixTalk listeners on one subnet are not receiving messages sent by talkers on a different subnet. What do I have to do?

Answer When communicating between subnets, the OrbixTalk multicast packets need to go through routers. In this case, you must ensure that those routers support multicast packets and that the multicast support is enabled on those routers. (This is specific to the particular router and you will need to consult the router's manual).

If you are doing this, then you should also be aware that OrbixTalk packets have a built-in time-to-live (TTL). This defaults to two (hops), so if you are intending for your messages to go through more than two routers, you need to increase the `IT_LIVE_TIME` parameter in your `Orbix.cfg` file to reflect the size of your network. The best way to check whether this is working correctly is to start up an OrbixTalk Directory Enquiries daemon (`otd/otdsm`) on each machine that you wish to communicate between. If these are all on the same

IT_DIRENQ_IPADDR and IT_DEFAULT_DIRENS_PORT, they should detect each other and all but the first to start will exit. If this does not happen, there is a problem with the network between the machines or with your configuration.

For more information about the IT_LIVE_TIME configuration parameter, refer to Appendix A, "Configuration Parameters".

Question: otd Daemons on Separate Subnets

When we run OrbixTalk Directory Enquiries daemons (otd/otdsm) on two separate subnets, can they share the $ORBIXTALKHOME directory?

The reason for the question is we have the same disk mounted on machines on different subnets. We start the otd/otdsm from two machines on different subnets sharing the OrbixTalk installation and hence the .dat files.

Answer

There should (in theory) be no problem doing this. The log files should not be corrupted by having more than one otd/otdsm reading/writing from/to it. This is definitely not advisable though, because it would involve having at least one of the otd/otdsms accessing the log files over an NFS connection. This is going to create an unnecessary and undesirable change in performance.

These files are accessed (read/written to) very regularly in a normal OrbixTalk system, and it is highly desirable to minimize the disk access time.

It is recommended that you have each otd with a different IT_CONFIG_PATH pointing to a different Orbix.cfg file. In each of the Orbix.cfg files, the IT_OTD_STORE and IT_APP_STORE parameters should point to a local directory path.

(The same applies to IT_MS_STORE_DIR and IT_MS_COMPACT_BACKUP_DIR in OTSFP).

For more information, refer to the Appendix A, "Configuration Parameters".

Question: Reducing Network Traffic

I've just had our network administrator complaining about the level of IP multicast packets being generated by my machine onto the network. Upon investigation we found that the traffic is being generated by the OrbixTalk

Directory Enquiries daemon (`otd/otdsm`). We are running it "straight out of the box". Can you give me any pointers about what to do to reduce the amount of traffic that this daemon is generating?

Answer

All OrbixTalk applications (including the `otd/otdsm` and `otmsd`) send out status messages as a part of the Reliable Multicast Protocol. You will notice these status messages, even when there are no messages being sent. The number and frequency of these messages can be configured easily.

The time-to-live (TTL) for these messages can also be adjusted to reduce their range and effect.

If your network administrator is concerned about extra traffic on a particular IP address or port, you can also configure the range of addresses that OrbixTalk uses. See the information about the `IT_DIRENQ_IPADDR`, `IT_DIRENQ_IPADDR_RANGE` and `IT_DEFAULT_DIRENS_PORT` configuration parameters in Appendix A, "Configuration Parameters".

Other configuration parameters which you may need to change are:

```
IT_DIRENQ_WILD_INTERVAL
IT_MAX_FT_HEART_BEAT
IT_INFO_INTERVAL
IT_INFO_COUNT
IT_NAK_RETRY
IT_NAK_RETRY_TIME
IT_FT_HEART_BEAT_INTERVAL
```

The following are only relevant for the Store and Forward Protocol (`otsfp`):

```
IT_MSG_MS_STATUS_INTERVAL
IT_MS_STATUS_RETRYS
IT_ACK_RETRY
IT_ACK_RETRY_TIME
```

For more information about the configuration parameters, refer to Appendix A, "Configuration Parameters".

Question: Are My Daemons Dead

How can I tell if my OrbixTalk Directory Enquiries Daemon (otd) and/or MessageStore Daemon (otmsd) have died?

Answer There are two ways of doing this.

Method One: Exception Handling

If any OrbixTalk talker or listener object attempts to contact either of these daemons but is unable to do so, an exception is raised. In general, if any OrbixTalk application is started without the otd running, you see the following exception raised (try it yourself by starting the rmpStockDemo talker without running the otd):

```
10098-- Communication failure
   - OrbixTalk error
   Reason: 53 - topic not bound
   [Completion status : COMPLETED_NO]
```

This exception is raised after maxRetries boot messages. maxRetries is the greater of the following two expressions:

```
1) IT_FT_HEART_BEAT_INTERVAL * IT_MAX_FT_HEART_BEAT
                              / IT_DIRENQ_INTERVAL
2) IT_DIRENQ_RETRYS
```

Also, if an OrbixTalk talker object attempts to talk on an OTSFP topic but is unable to contact an otmsd, the following exception is raised (you can try it yourself by starting an sfpStockDemo talker with an otd, but no otmsd):

```
10098-- Communication failure
   - OrbixTalk error
   Reason: 54 - message store not known
   [Completion status : COMPLETED_NO]
```

This exception is raised after IT_ACK_RETRY un-acknowledged resynchronization requests are sent by the talker object to the otmsd at an interval of IT_ACK_RETRY_TIME milliseconds.

If your otmsd should die at some time after the talker object has begun talking, you will see the following exception the next time a talk is attempted (again, try it yourself by killing the otmsd while running the sfpStockDemo):

```
   10098-- Communication failure
```

```
- OrbixTalk error
Reason: 61 - message timed out
[Completion status : COMPLETED_NO]
```

As your OrbixTalk application cannot function in any meaningful capacity without the daemons, you must catch and act upon these exceptions.

When you catch one of these exceptions, your application should attempt to restart the missing daemon(s) automatically by executing a shell command, or pass a meaningful error message on to the user.

Method Two: The OrbixTalk Daemon Process Detection Tool (otpsd)

The `otpsd` is a tool supplied with your OrbixTalk installation.

You can use this tool to check whether the requisite daemons are available before attempting to register any OrbixTalk talker or listener objects.

This tool can be run at any point in a shell command, and will output information in a format similar to the following:

```
Identifying running daemons (IP Addr=225.0.0.0, Port=5000)...
Completed detection phase.
Detected the following daemons :
XXX.XXX.XXX.XXX  myhost.iona.com      otmsd
//OrbixTalk/MessageStore1
XXX.XXX.XXX.XXX  myhost.iona.com      otmsd
//OrbixTalk/MessageStore2
XXX.XXX.XXX.XXX  myhost.iona.com      otd
```

This indicates that within the OrbixTalk system, which is running on `IT_DEFAULT_DIRENS_PORT = 5000` and `IT_DIRENQ_IPADDR = 225.0.0.0`, there are two MessageStore daemons and one Directory Enquiries Daemon running. One `otmsd` has `IT_MS_TOPIC = //OrbixTalk/MessageStore1`, and the other has `IT_MS_TOPIC = //OrbixTalk/MessageStore2`.

Your application can parse the output from the `otpsd` tool to confirm that everything is running correctly before starting any talker or listener processes.

A typical OrbixTalk system can incorporate a combination of these two approaches.

Question: Slow Store and Forward System

Why does my Store and Forward (SFP) system run so slow?

Answer The Store and Forward Protocol (SFP) is always slower than a Reliable Multicast Protocol (RMP) or raw Multicast Protocol (MCP) system sending the same message. There are several reasons for this:

- Messages must be written to disk.
- The message store process provides a single bottle-neck that all messages pass through.
- Listener processes must maintain their state by writing it to disk.

Given these points there are some basic things you can do to ensure that your system runs as fast as possible:

- Ensure that the message store database is on a disk directly connected to the machine running the message store. Using a database on an NFS mounted disk can drastically reduce throughput; 25% throughput is typical.
- Have a dedicated machine to support the message store process. If the message store is contending with other processes for disk write time and processor time it will suffer. Listeners are particularly bad for this because they do both.
- Although an application can only use a single message store, there is no reason to limit your system to a single message store. If you can partition your system such that applications use distinct sets of topics, then consider using multiple message stores.

Question: Compiling OrbixTalk Code

When I compile my OrbixTalk code as a DLL, it doesn't seem to work. What am I doing wrong?

Answer This problem can occur if you use the "multithreaded" runtime library rather than the "multithreaded DLL" runtime library. You can set this option in Microsoft Developer using the Project Settings dialog, C/C++ Tab, Code Generation category. Alternatively, you can just specify `/MD` rather than `/MT`.

Question: otd Daemon Shutdown

Why does my OrbixTalk Daemon (`otd`) shutdown immediately when I try to start it?

Answer When I try to start the `otd` from the command line I get a message saying that its `Starting as Secondary`. But soon after I get a message saying `Primary already running. Exiting...`. What is happening?

This means that there is already an OrbixTalk daemon `otd` running on this IP multicast address range on this network.

An OrbixTalk system corresponds to a range of IP multicast addresses on a network, and there can only be one `otd` running in each OrbixTalk system. All OrbixTalk applications in that system use the same OrbixTalk daemon.

Question: Communicating Across Subnets

My listeners/talkers cannot communicate across subnets, even though I have routers that support multicast. Why?

Answer When communicating between subnets, your OrbixTalk Multicast packets must pass through routers. In this case, you must ensure that those routers support multicast packets (most do) and that the multicast support is enabled on those routers. (This is specific to the particular router and you will need to consult the router's manual).

If you are doing this, then you should also be aware that OrbixTalk packets have a built-in time-to-live (TTL). This defaults to two (hops) so if you intend your messages to go through more than two routers, then you must increase the `IT_LIVE_TIME` parameter in your `iona.cfg` file to reflect the size of your network.

The best way to check whether this is working correctly is to start an `otd` on each machine that you want to communicate. If these daemons are all on the same `IT_DIRENQ_IPADDR` and `IT_DEFAULT_DIRENS_PORT`, they should detect each other. All but the first to start will exit.

If this does not happen, there is a problem with the network between the machines or with your configuration.

Question: Talking to Different Machines

OrbixTalk works on one machine, but I cannot get it to talk to a different machine.

Answer This situation can occur, for example, when you install a listener on a PC and a talker on a Sun machine.

- Set the `IT_LOCAL_DOMAIN` to null (that is, leave the rest of the line in your `iona.cfg` file blank.)
- Check that the `IT_DIRENQ_IPADDR`, `IT_DIRENQ_IPADDR_RANGE`, `IT_DEFAULT_DIRENS_PORT` and `IT_DIRENS_NAME` parameters are the same on the Sun machine as on the PC.

These points should allow your machines to communicate if they are on the same subnet. If not, ensure that any routers between the two subnets support multicast, and that this functionality is enabled.

If these settings are correct, you can test the system by attempting to start an `otd` on both machines. One `otd` should go primary and the other should exit. If not, your network may not be properly configured for Multicast. Try `ifconfig -a` on Solaris to check your network interface.

Question: Multiple OrbixTalk Systems

Can I have more than one OrbixTalk system running on my subnet?

Answer Requirement:

This can arise, for example, if you have different developers working on OrbixTalk applications in parallel. They each want to have control over their own `otd` and need to ensure that other tests do not interfere with theirs.

Solution:

It is possible to have more than one OrbixTalk system running on a single subnet, provided that the following precautions are taken:

1. Each system must operate on a different (non-overlapping) IP Address range. You will probably already have noticed that if you start more than one `otd`, the second exits under normal circumstances. To achieve this,

you must set the IT_DIRENQ_IPADDR and IT_DIRENQ_IPADDR_RANGE parameters in the orbixtalk3.cfg file used by each otd (pointed to by IT_CONFIG_PATH) so that there are no overlaps.

IT_DIRENQ_IPADDR is the multicast IP address used to communicate with the otd. As long as each otd uses a different IP address, you can have multiple otds running in the subnet.

IT_DIRENQ_IPADDR_RANGE usually corresponds to the maximum hardware limit for the machine in use.This is really the number of multicast groups that the ethernet card can join. If you want to run more than one OrbixTalk system on a single machine, you must divide this range to allow each system to have some access to these groups.

2. Ensure that each system has unique values for the following configuration pairs:

   ```
   (IT_MS_STORE_DIR, IT_MS_STORE)
   (IT_OTD_STORE, IT_OTD_APPLISTORE)
   (IT_LINK_STORE, IT_OTD_LINKSTORE)
   (IT_OTD_STORE, IT_OTD_TOPICSTORE)
   ```

   If these pairs are not unique, more than one system can end up writing different IP addresses to the same set of statelogs.

3. Be sure to clean up all the statelog `.dat` and `.ndx` files whenever you make changes to the configuration, particularly the IP address ranges.

Part VI

Appendices

Appendix A
Configuration Parameters

This appendix provides information about the general configuration parameters that are used with OrbixTalk.

Overview

This chapter provides the following:

- An alphabetical list of the general configuration parameters that are used with OrbixTalk. The configuration parameters used by the IIOP Gateway are detailed in Appendix B, "IIOP Gateway Configuration Settings".
- Detailed information about each parameter and how the parameters affect each other. The parameters are divided into the following categories:
  - Reliable Multicast Protocol.
  - Store and Forward Protocol.
  - Directory Enquiries daemon.
  - Network.
  - General.

Alphabetical List of Configuration Parameters

The following table provides an alphabetical list of the configuration parameters for OrbixTalk.

Configuration Parameter	Description
IT_ACK_RETRY	Number of otsfp message transmission retries before reporting an exception. Default Value: 3. For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_ACK_RETRY_TIME	Time between successive otsfp message transmission retries. Default Value: 5000 (ms). For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_APP_STORE	The IT_APP_STORE parameter sets the directory in which OTSFP consumer application state logs are stored. For example, when you run the sfpStock demo consumer, a file named stocksfp-listener-Listen.dat is created in the directory that is pointed to by IT_APP_STORE. If you change this directory you do not change the directory where the otd stores its application information (AppliStore.* and TopicStore.*). If you want to change where these files are created, you should change the IT_OTD_STORE parameter. Default Value: . (current directory). For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_BASE_FRAG_WINDOW_SIZE	Number of missing message fragments before message requests (NAKs) are sent to the originating supplier application. Default Value: 10 (number of message fragments). For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_BATCH_SIZE	Number of fragments per batch (see also IT_MIN_BATCH_INTERVAL) Default value: 10. For more information, refer to "Flow Control Mechanism Configuration Parameters" on page 189.
IT_DEFAULT_DIRENS_PORT	Directory Enquiries port used for all communications with the otd process. Default Value: 5000. For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description

IT_DIRENQ_INTERVAL	Determines the time between attempts to connect to the otd (see IT_DIRENQ_RETRYS).
	Default Value: 5000 (ms).
	The otd requires the exclusive use of two ports to operate properly. The port numbers used are set using the IT_DEFAULT_DIRENS_PORT parameter. The otd uses IT_DEFAULT_DIRENS_PORT and IT_DEFAULT_DIRENS_PORT + 1). If some other application is currently using one or more of the UDP ports required by the otd, you can see this error:
	`mmm dd hh:mm:ss otd: ERROR!: Failed to bind to x.x.x.x:port#`
	`mmm dd hh:mm:ss otd: ERROR!: FATAL error - Failed to open socket, probably not enough file descriptors`

IT_DIRENQ_IPADDR	Multicast IP Address used by OrbixTalk processes for all communications with the otd. Default Value: 225.0.0.0 For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.
IT_DIRENQ_IPADDR_RANGE	Determines the maximum number of addresses that the otd can allocate. Default Value: 10 for HPUX10.x; 31 for all other platforms. For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_DIRENQ_RETRYS	Determines the number of retries made to the otd before the connect attempt to the otd fails. Default Value: 6. For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.
IT_DIRENQ_WILD_INTERVAL	Determines the interval between each PROD message. Default Value: 18,000,000 (ms). For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.
IT_DIRENS_NAME	Name used by all OrbixTalk processes to communicate with the otd. Default Value: //OrbixTalk/DirectoryEnquiries For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.
IT_FRAG_ACCELERATE	Determines when to decrease the current interval between transmission of each fragment. Default Value: 1000 (ms). For more information, refer to "Flow Control Mechanism Configuration Parameters" on page 189.
IT_FRAG_INTERVAL	Initial interval between message fragments (ms). Default Value: 50 (ms). For more information, refer to "Flow Control Mechanism Configuration Parameters" on page 189.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_FRAG_WARP_DRIVE	Point at which the fragment transmission lower interval limit is decreased. Default Value: 5. For more information, refer to "Flow Control Mechanism Configuration Parameters" on page 189.
IT_FT_HEART_BEAT_INTERVAL	The frequency at which otd and otmsd "ping" (in ms). Default Value: 1000 (ms). Forced minimum of 250 ms. For more information, refer to "Fault Tolerance Configuration Parameters" on page 204.
IT_INFO_COUNT	The number of information messages that are sent after a pause in transmission of messages. Default Value: 5. For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_INFO_INTERVAL	The time between successive information messages for otrmp. Default Value: 2000 (ms). For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_LINK_STORE	Directory containing the Directory Enquiries Link Store database. Default value: . (current directory). For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_LIVE_TIME	Time to live for message fragments (packets) multicast on the network Default Value: 2. For more information, refer to "Network Configuration Parameters" on page 201.
IT_LOG_CONSOLE	If the Console output flag is 1, OrbixTalk processes output text messages to the console. Default Value: 1. For more information, refer to "General Configuration Parameters" on page 202.
IT_LOG_FLAGS	Sets individual logging output levels. Default Value: "" For more information, refer to "General Configuration Parameters" on page 202.
IT_LOG_LEVEL	The system logging output level from 0 to 32. Default value: 0. For more information, refer to "General Configuration Parameters" on page 202.
IT_LOG_SYSLOG	If the SYSLOG output flag is 1, OrbixTalk processes output text messages to a file rather than the standard output. Default Value: 0. For more information, refer to "General Configuration Parameters" on page 202.
IT_LOG_TID	Adds the thread ID to logging output when set to 1. Default Value: 0.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_MAX_ACK_KB	Maximum size for outstanding information before an acknowledgement is received. Default value: 1280 Kilobytes. For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_MAX_FRAG_INTERVAL	Maximum interval between message fragments (ms). Default Value: 500 (ms). For more information, refer to "Flow Control Mechanism Configuration Parameters" on page 189.
IT_MAX_FT_HEART_BEAT	Number of heartbeats that are missed before the otd/otmsd determines that no other daemon is running and becomes primary. Default Value: 6. Forced to minimum of 10 if Fault Tolerance is enabled. For more information, refer to "Fault Tolerance Configuration Parameters" on page 204.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_MAX_MSG_SIZE_KB	Maximum size of the messages that can be sent by your OrbixTalk application, in Kilobytes. The size of the user data which can be sent is less than the value that is set in IT_MAX_MSG_SIZE_KB. The reason for this is that OrbixTalk adds some header information to each message. The amount of this header information varies depending upon the protocol being used, and the degree of fragmentation of the message. The header information is typically around 150 bytes per OTRMP message. Default value: 200 Kilobytes. For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_MAX_PEND_KB	Size limit of the pending message queue. Default Value: 1280 (Kilobytes) For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_MAX_RECV_KB	The maximum number of Kilobytes queued at the receiver. Default Value: 1280 (Kilobytes). For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_MAX_SENT_KB	The maximum number of Kilobytes retained in the message queue. Default Value: 1280 (Kilobytes). For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_MAX_SENT_TIME	The maximum length of time that message fragments are stored. Default Value: 30000 (ms). For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_MC_INTERFACE	Multicast Interface IP address. Default value: 0.0.0.0 For more information, refer to "Network Configuration Parameters" on page 201.
IT_MIN_BATCH_INTERVAL	Minimum batch interval. Default value: 10. For more information, refer to "Flow Control Mechanism Configuration Parameters" on page 189.
IT_MS_COMPACT_BACKUP_DIR	Before compaction, the existing MessageStore is copied to this location. Default Value: . (current directory). For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_MS_COMPACT_BATCH_SIZE	Determines the number of Kilobytes to be copied at each interval (see IT_MS_COMPACT_INTERVAL). Default Value: 300. For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_MS_COMPACT_INTERVAL	Frequency of compaction operations. Default Value: 3000 (ms). For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_MS_STATUS_RETRYS	Number of status messages sent by the otmsd to listener applications. Default Value: 3. For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_MS_STATUS_TOPIC_LIVE_SECS	If no messages are received within this time period, the topic is deleted. Default Value: 900 (secs). For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_MS_STORE	The MessageStore name used by otmsd to store its messages. Default Value: MsgStore. For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_MS_STORE_DIR	The MessageStore directory path. Default Value: . (current directory) For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_MS_TOPIC	MessageStore topic used by otmsd and all Store and Forward (SFP) processes that communicate with otmsd. Default Value: //OrbixTalk//MessageStore For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_MSG_MS_STATUS_INTERVAL	Time between transmission of status messages from the otmsd to listener applications. Default Value: 11,000 (ms). For more information, refer to "Store and Forward Protocol Configuration Parameters" on page 191.
IT_NAK_RETRY	Maximum number of Negative Acknowledgements (NAKs) sent from a consumer application. Default Value: 3. For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_NAK_RETRY_TIME	Time interval between requests for a supplier application to resend a message. Default Value: 1000 (ms). For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_OT_DAEMON_BACKGROUND	Causes OrbixTalk daemons to start as background processes. Default Value: 0 For more information, refer to Chapter 13 "Daemons".
IT_OT_FT_ENABLE	Enables Fault Tolerance support in all OrbixTalk daemons. Default Value: 0 For more information, refer to "General Configuration Parameters" on page 202.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_OTD_APPLISTORE	Holds Application Names used by OrbixTalk processes. Default Value: AppliStore For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.
IT_OTD_LINKSTORE	Stores associations between applications and the topics they are registered for talking/listening on. Default value: LinkStore For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.
IT_OTD_STORE	Directory path for the Application and Topic Stores (written to by the otd). Default Value: . (current directory) For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.
IT_OTD_TOPICSTORE	Holds all the Topic Name-to-IP Address mappings. Default Value: TopicStore For more information, refer to "Directory Enquiries Daemon Configuration Parameters" on page 196.
IT_RECV_SOCKET_BUFF_SIZE	Default recv Socket buffer size. Default Value: 65536 (bytes) For more information, refer to "Network Configuration Parameters" on page 201.

Table 17.1: *Configuration Parameters*

Configuration Parameter	Description
IT_SEND_FRAG_WINDOW_SIZE	Number of message fragments determining when the rate of transmission of message fragments should be reduced. Default Value: 100 (number of message fragments). For more information, refer to "Reliable Multicast Protocol Configuration Parameters" on page 183.
IT_SEND_SOCKET_BUFF_SIZE	Default send Socket buffer size. Default Value: 65536 (bytes) For more information, refer to "Network Configuration Parameters" on page 201.
IT_TALK_PEND_INTERVAL	Interval between batches. Default value: 50 (ms) For more information, refer to "Flow Control Mechanism Configuration Parameters" on page 189.

Table 17.1: *Configuration Parameters*

Detailed Information about the Configuration Parameters

This section describes the configuration parameters for OrbixTalk in more detail. OrbixTalk provides default values for all configurable elements; however, you may need to configure these values for your particular environment.

The OrbixTalk configuration variables are contained within a scoped configuration file called `orbixtalk3.cfg`.

OrbixTalk variables can be modified using the Orbix Configuration Explorer. Please refer to the *Orbix C++ Administrator's Guide* for details on using the Orbix Configuration Explorer.

All databases maintained by OrbixTalk applications consist of two files. The first is a file with a ".dat" extension which indicates it is a data file. The second file has a ".ndx" extension and is an index file to the ".dat" file. These files are maintained by the following OrbixTalk processes: the OrbixTalk Directory

Enquiries daemon (otd/otdsm) and the OrbixTalk MessageStore daemon (otmsd). The state log files maintained by consumer applications using the Store and Forward Protocol (otsfp) do not have a ".ndx" file.

The configuration parameters available in OrbixTalk are listed in the following categories:

- Reliable Multicast Protocol
- Store and Forward Protocol
- Directory Enquiries daemon
- Network
- General

Note: Some of the configuration parameters contain a warning. Modify these configuration variables with caution as they can adversely affect the communications between OrbixTalk processes.

Viewing Current Configuration Settings

To view a short description of each configuration parameter and its current setting, run the OrbixTalk Directory Enquiries daemon with the -v flag as follows:

```
otd -v
```

Setting Configuration Parameters

To modify the configuration parameters, do one of the following:

- Change the setting in the orbixtalk3.cfg file that is included within the file pointed to by the IT_IONA_CONFIG_FILE or IT_IONA_CONFIG_PATH.
- Change the environment variable with the same name as the parameter.

For example, to modify the setting for the base IP multicast address (232.0.0.50), you can either add the following entry to the orbixtalk3.cfg file:

```
IT_DIRENQ_IPADDR 232.0.0.50
```

or perform the following command at the shell prompt (the actual command depends on the shell being used):

```
export IT_DIRENQ_IPADDR=232.0.0.50
```

Where a setting exists in the `orbixtalk3.cfg` file and as an environment variable, the environment variable value takes precedence over the setting in the configuration file.

Using Multiple Configuration Settings

In general, all applications in an OrbixTalk system communicate with one OrbixTalk Directory Enquiries daemon (`otd`) so that there is a consistent mapping between Topic Names and Topic IDs. This requires all applications to have the same settings for the parameters related to the OrbixTalk Directory Enquiries daemon.

When specific applications in an OrbixTalk system require different configuration settings to those used by other applications, you need to do one of the following:

- In the shell from which the specific applications are started, change the required configuration parameters using environment variables.
- Use multiple `orbixtalk3.cfg` files and change the `IT_CONFIG_PATH` environment variable to point to the appropriate directory.

For example, when using the Store and Forward protocol, all communication between suppliers and consumers takes place through an intermediary process—the MessageStore.

Using Multiple MessageStore Daemons

An OrbixTalk system normally uses a single MessageStore daemon for all communication with the `otsfp` protocol. However, OrbixTalk can be configured to use multiple MessageStore daemons.

Each MessageStore daemon must use a separate MessageStore topic. However, there is only one configuration parameter to set the MessageStore topic in each `orbixtalk3.cfg` file. Therefore, each MessageStore daemon, and the set of applications that use that daemon, must use a separate `orbixtalk3.cfg` file.

Create multiple orbixtalk3.cfg files, each with a different value for the IT_MS_TOPIC parameter, and change the IT_CONFIG_PATH environment variable in the shell from which each OrbixTalk application is started.

For example, a system that uses two MessageStore daemons, one for the otsfp//A topic, and one for the otsfp//B topic, has two orbixtalk3.cfg files:

- orbixtalk3.cfg (A) which contains IT_MS_TOPIC //MessageStore/A and is placed in the /OrbixTalk/ConfigA directory.
- orbixtalk3.cfg (B) which contains IT_MS_TOPIC //MessageStore/B and is placed in the /OrbixTalk/ConfigB directory.

All other settings remain the same in these files.

There are two sets of processes in the system, one of which is interested only in topic otsfp//A, and the other only in topic otsfp//B.

A single OrbixTalk Directory Enquiries daemon is started from a shell with IT_CONFIG_PATH set to /OrbixTalk/ConfigA. The first MessageStore daemon is started from a shell with the IT_CONFIG_PATH environment variable set to /OrbixTalk/ConfigA, and the other MessageStore daemon is started from a shell where IT_CONFIG_PATH is set to /OrbixTalk/ConfigB.

Processes that use the otsfp//A topic must be started from a shell that references the orbixtalk3.cfg (Config A) file, while processes using the otsfp//B topic must be started from a shell that uses the orbixtalk3.cfg (Config B) file. All processes use the same OrbixTalk Directory Enquiries daemon, but the two MessageStore daemons service only the topics used by those processes that access the same orbixtalk3.cfg as the specified MessageStore daemon.

Reliable Multicast Protocol Configuration Parameters

This section discusses the configuration parameters that affect the performance of the Reliable Multicast Protocol.

The OrbixTalk Reliable Multicast Protocol (otrmp) is used for communication between suppliers and consumers in an OrbixTalk system that does not use the MessageStore. It achieves reliability and scalability with a negative acknowledgement (NAK) scheme. Rather than each supplier ensuring that all

messages are received by all consumers, it is the responsibility of individual consumers to ensure that each supplier sends it all the messages in which it is interested.

Messages sent by suppliers are divided into fragments. Each fragment is sent in sequence, and has an associated sequence number. Once a fragment has been sent by a supplier, it is added to a queue of sent fragments.

Consumers that receive fragments can determine if a fragment has been missed using the sequence numbers attached to each fragment. When a fragment is lost, a consumer sends a negative acknowledgement (NAK) to the supplier that sent the fragment, requesting that the fragment is re-sent. If the fragment remains in the supplier's queue of sent fragments, the supplier re-sends the fragment.

The Reliable Multicast Protocol uses two other mechanisms to achieve reliability:

- INFO messages.

  INFO messages are sent by suppliers when there is a period of inactivity in sending messages. An INFO message is sent on each topic, specifying the sequence number of the most recently sent message fragment. Consumers that receive INFO messages use them to determine if any fragments have been lost. If a supplier stops sending messages, the consumers still need to determine if any fragments have been missed.

- Flow control applied to suppliers.

  If all suppliers send messages as quickly as possible, consumers are overwhelmed with the arrival of messages and are unable to process them. The flow control mechanism of the Reliable Multicast Protocol ensures that suppliers send messages at a rate no faster than consumers can receive them and attempts to keep this rate as high as possible.

  Each message that is sent by a supplier is placed in a queue of pending messages before it is sent on the network. The flow control is applied to this queue, releasing fragments from the queue at a steady rate related to the number of NAKs received recently and the history of message transmission.

Normally, a remote method invocation using OrbixTalk returns immediately because the communication is asynchronous. If the pending message queue is full because the user is attempting to send messages faster than the limits imposed by the flow control mechanism, the remote method invocation blocks until space in the pending message queue becomes available.

IT_MAX_MSG_SIZE_KB	Default value: 2560 Kilobytes

The Reliable Multicast Protocol imposes a limit on the size of a message that can be sent. The limit is set by the IT_MAX_MSG_SIZE_KB configuration parameter. If you attempt to send a message that exceeds this limit, an exception is raised, and the message is not sent.

IT_INFO_INTERVAL	Default value: 2000 (ms)

INFO messages are sent at regular intervals by OrbixTalk applications for each topic on which messages are being sent. INFO messages ensure that listening applications (which can also be otd or otmsd) have not missed messages. The IT_INFO_INTERVAL configuration parameter specifies the interval in milliseconds between each INFO message.

IT_INFO_COUNT	Default value: 5

This count determines the number of information messages that are sent by a supplier, for each topic, after a pause in the transmission of messages. Note that otrmp is used by otsfp so this parameter affects both protocols.

In combination with the IT_INFO_INTERVAL parameter, this parameter determines the period of time over which INFO messages are sent and, therefore, the maximum time of communication failure for which the otrmp protocol can be considered reliable.

IT_NAK_RETRY	Default value: 3
IT_NAK_RETRY_TIME	Default value: 1000 (ms)

When using the Reliable Multicast Protocol, consumers send negative acknowledgements (NAKs) to determine when a message fragment has not been received. If the message is not received by the consumer within a period of time

equal to IT_NAK_RETRY_TIME, the NAK is sent again. The IT_NAK_RETRY parameter specifies the number of times a NAK is sent by a consumer before it raises an exception to indicate communication failure.

If a fragment of a message fails to be received within the IT_NAK_RETRY_TIME interval, a message request is sent from the consumer application to the supplier application requesting that the message is resent.

IT_BASE_FRAG_WINDOW_SIZE	Default value: 10 (number of message fragments) **WARNING**: Modify this parameter with caution because changes can drastically affect performance of the protocols if not tuned to the environment.
IT_SEND_FRAG_WINDOW_SIZE	Default value: 100 (number of message fragments) **WARNING**: Modify this parameter with caution because changes can drastically affect performance of the protocols if not tuned to the environment.

Base fragment window size determines the size of the sliding window used to detect when message requests (NAKs) are required to be sent for missing fragments. If missing fragments exist outside this window, message requests (NAKs) are sent to the originating supplier application (which can also be otmsd).

In a high throughput situation, IT_BASE_FRAG_WINDOW_SIZE defines the number of fragments that can be received by a consumer before a retry is sent. This overrides IT_NAK_RETRY_TIME when fragments are received very quickly.

IT_SEND_FRAG_WINDOW_SIZE defines the size of the sliding window used by supplier applications and the OrbixTalk MessageStore daemon to detect when the rate of transmission of message fragments should be reduced. If any message requests (NAKs) are received outside this window, the flow control mechanism lowers the rate of transmission.

IT_SEND_FRAG_WINDOW_SIZE defines the maximum difference allowed between the most recent fragment sent by a supplier and one indicated in a NAK before flow control is applied.

IT_MAX_SENT_KB	Default value:1280 Kilobytes
IT_MAX_SENT_TIME	Default value: 30000 (ms)
IT_MAX_RECV_KB	Default value:1280 Kilobytes

Talking applications retain the message fragments they have sent in case a consumer requests them to be resent. The IT_MAX_SENT_KB parameter sets the maximum number of Kilobytes of the queue of sent messages (consumer application or OrbixTalk MessageStore daemon). Fragments are stored in this buffer until contiguous regions are formed and can be copied to form all, or part, of a message. When fragments are frequently missed, this variable can be increased. However, the default value is sufficient in most circumstances.

The IT_MAX_SENT_TIME parameter specifies the amount of time for which they are retained for retries (performed on receipt of an RMP message request/NAK). This applies to supplier applications using otrmp/otsfp, including the OrbixTalk MessageStore daemon.

IT_MAX_SENT_KB specifies the upper limit to the sent message queue size in Kilobytes. Once the limit is reached, the oldest fragments are removed to make space as required for new message fragments to be added to the queue.

IT_MAX_SENT_TIME specifies the time in milliseconds that message fragments remain in the sent message queue. Once the time has expired, they are removed from the queue.

When a consumer application is executing code other than the Orbix event loop, OrbixTalk messages arrive and are queued at the receiver. They are not dispatched to the appropriate user function until Orbix processes events in its event loop. The limit to the amount of data that is queued at a consumer is set by the IT_MAX_RECV_KB parameter, which specifies the size of the receiving queue in Kilobytes.

In combination, the IT_MAX_SENT_KB, IT_MAX_SENT_TIME and IT_MAX_RECV_KB parameters affect the amount of time over which a supplier is guaranteed to have messages available to be resent to a consumer that requests them with a NAK. The following example shows a situation in which it is necessary to modify the parameters to achieve a maximum rate of reliable message delivery.

For example, an OrbixTalk system that consists of a single supplier, talking 128 Kilobyte messages once every 5 seconds, and a single consumer that does not use a thread filter so each message is dispatched in a sequential manner.

The amount of time taken to process each message in user code can vary between 1 and 10 seconds, with the following probability:

Time taken to process message	Probability
1 second	0.1

5 seconds	0.8
10 seconds	0.1

In this system, there is a probability that the incoming message queue will overflow, because of the distribution of processing times. This will result in the supplier being slowed down by the flow control mechanism. However, as the average rates of sending and receiving messages are equal, this can be minimized by increasing the queue size limit at the consumer.

`IT_MAX_PEND_KB` Default value: 1280 Kilobytes

The `IT_MAX_PEND_KB` configuration parameter sets the size limit of the pending message queue. Messages are added to the pending queue by a user invocation of a remote method or a `push()` operation. Messages are removed from the queue by the flow control mechanism as it sends message fragments. Once the queue becomes full, remote method invocations block until space becomes available in the queue.

Increase the size of this parameter when invocations are being made rapidly as queue becomes full in periods; for example, when large messages are sent or when the rate at which remote method invocations are made is variable.

Flow Control Mechanism Configuration Parameters

IT_TALK_PEND_INTERVAL	Interval between batches. Default value: 50 (ms)
IT_FRAG_INTERVAL	Initial fragment interval (ms). Default Value: 50 (ms)
IT_MAX_FRAG_INTERVAL	Maximum fragment interval (ms). Default Value: 500 (ms)
IT_FRAG_ACCELERATE	Point at which to decrease interval. Default Value: 1000 (ms) **WARNING**: Modify this parameter with caution because changes can drastically affect performance of the protocols if not tuned to the environment.
IT_FRAG_WARP_DRIVE	Point at which the fragment transmission lower interval limit is decreased. Default Value: 5 **WARNING**: Modify this parameter with caution because changes can drastically affect performance of the protocols if not tuned to the environment.
IT_MIN_BATCH_INTERVAL	Minimum batch interval. Default value: 10
IT_BATCH_SIZE	Number of fragments per batch. Default value: 10

The IT_FRAG_INTERVAL parameter determines the initial fragment interval. The flow control mechanism can change this initial interval once transmission begins.

The IT_MAX_FRAG_INTERVAL parameter determines the maximum interval between message fragments (ms). The flow control mechanism will not exceed this value.

The IT_FRAG_ACCELERATE parameter determines when to decrease the current interval between the transmission of each fragment. If there are no message requests (NAKs) received in this time, the interval is decreased so that message fragments are output at a faster rate.

The IT_FRAG_WARP_DRIVE parameter controls the point at which the fragment transmission lower interval limit is decreased. During transmission, a lower limit is set on the interval between transmission of fragments depending on the rate at which consumers can receive messages. If the slowest consumer disappears the rate of transmission can be increased. This parameter determines how many times an attempt to increase the rate of transmission of message fragments is made before the interval between fragment transmission is further decreased. An increase in the rate of transmission of message fragments can only occur if no incoming message requests/NAKs are received in the mentioned period.

The IT_MIN_BATCH_INTERVAL parameter determines the time interval between batches of fragments.

Message fragments are taken from the pending message queue and sent on the network by the flow control mechanism. This set of parameters determines the way in which flow control is applied to message transmission. Flow control is applied on a per-topic basis, and activity on one topic does not affect the flow control applied to another.

Fragments are sent in batches, with a period of time between batches that limits the rate at which fragments are sent. The configuration parameters enable you to set:

- limits to the rate of message transmission
- the rate at which the message transmission rate is modified (acceleration)
- the rate at which acceleration rate is modified

The period between batches varies between IT_MAX_FRAG_INTERVAL and IT_MIN_BATCH_INTERVAL starting from IT_FRAG_INTERVAL. When a consumer requests a slowdown, or sends a NAK for a very old fragment, the interval is increased to enable the consumer to catch up. If no NAKs or slowdown requests have been received during the interval specified by the IT_FRAG_ACCELERATE parameter, the interval is decreased.

Flow control can be turned off by setting IT_FRAG_INTERVAL, IT_MAX_FRAG_INTERVAL and IT_MIN_BATCH_INTERVAL to the same value.

Store and Forward Protocol Configuration Parameters

The OrbixTalk MessageStore daemon (otmsd) acts as a supplier and consumer application. Some of the configuration parameters that modify low-level protocol parameters also affect the OrbixTalk MessageStore daemon as well as user-defined supplier and consumer applications. However, the OrbixTalk Directory Enquiries daemon uses its own Directory Enquiries Protocol (DEP) so parameters that affect supplier and/or consumer applications generally have no affect on the OrbixTalk Directory Enquiries daemon.

Individual processes, such as the OrbixTalk Directory Enquiries daemon and the OrbixTalk MessageStore daemon, can be set up with their own environment and orbixtalk3.cfg file. This resolves such issues as sharing configuration parameters and environment variables

IT_MS_STORE	Message store name. Default Value: MsgStore
IT_MS_STORE_DIR	Message store directory. Default Value: . (current directory)

The MessageStore daemon (otmsd) maintains a disk-based database. The IT_MS_STORE configuration parameter specifies the name of the files used.

Two files are created for every database used; one file has a .dat extension, and the other file has a .ndx extension. For example, if IT_MS_STORE is set to MessageStore, the two files used have the names MessageStore.dat, and MessageStore.ndx.

These files are created and accessed in the directory specified by the IT_MS_STORE_DIR configuration parameter. This configuration parameter specifies a directory that exists, and that has read and write permission for the user ID under which the MessageStore daemon is executed. Modify the IT_MS_STORE_DIR parameter to change the message store directory.

When using multiple MessageStore daemons in a single system, separate the database by specifying a different IT_MS_STORE_DIR parameter for each MessageStore daemon.

IT_MS_TOPIC	Message Store topic. Default Value: //OrbixTalk//MessageStore

When using the OrbixTalk Store and Forward Protocol (otsfp), suppliers send all messages to the MessageStore daemon on the MessageStore topic. The MessageStore daemon is responsible for forwarding these messages to consumers interested in them.

The IT_MS_TOPIC configuration parameter sets the name of the topic used by the MessageStore. It must be in a form similar to:

```
//OrbixTalk/MessageStore
```

Note: You should not specify a protocol before the opening // of the Topic Name. Specifying an invalid Topic Name results in an error message being produced by the MessageStore daemon on startup.

An OrbixTalk system can use multiple MessageStore daemons by specifying a distinct MessageStore topic for each MessageStore daemon. The daemon with which an application communicates is determined by this parameter. There is no communication between the different MessageStore daemons.

To run separate OrbixTalk MessageStore daemons, each environment must have different values for IT_MS_TOPIC, IT_MS_STORE and IT_MS_STORE_DIR to ensure that each MessageStore is unique.

IT_MS_COMPACT_BACKUP_DIR	Default value: . (The directory from which the MessageStore process was started)

On performing compaction, the MessageStore creates a set of temporary files. The IT_MS_COMPACT_BACKUP_DIR parameter specifies the directory into which those files are placed during compaction.

IT_MSG_MS_STATUS_INTERVAL	Default value: 11000 (ms)

The OrbixTalk Store and Forward Protocol (otsfp) guarantees message delivery using the MessageStore daemon to forward messages to consumers. Suppliers send messages on user-defined topics to the MessageStore, which sends an acknowledgment to the supplier for each message received. The MessageStore daemon then stores the messages in a disk-based database before forwarding the messages onto the consumers.

Guaranteed delivery is achieved through the use of STATUS messages, which are sent by the MessageStore to consumers similar to the INFO messages used by the Reliable Multicast Protocol (otrmp). Each STATUS message specifies the sequence number and application ID of the most recent message sent on each topic.

Consumers interested in a particular topic using the Store and Forward Protocol (otsfp) receive STATUS messages and are able to send replay requests for any messages that it has missed. The MessageStore records all messages in a disk-based database, so the messages are always available to be replayed.

The IT_MSG_MS_STATUS_INTERVAL parameter specifies the time between transmission of status messages from the MessageStore to consumer applications. Status messages enable a consumer to determine if it must request replays from the MessageStore. A status message contains the last transmitted otsfp sequence number.

In combination with IT_MS_STATUS_RETRYS, the IT_MSG_MS_STATUS_INTERVAL parameter represents the interval of communication failure over which the otsfp protocol maintains guaranteed delivery.

Note: It is important that the value of the IT_MSG_MS_STATUS_INTERVAL parameter is greater than the multiple of the Reliable Multicast Protocol INFO message interval (IT_INFO_INTERVAL) and INFO message count (IT_INFO_COUNT**).**

IT_MS_STATUS_RETRYS Default value: 3

The IT_MS_STATUS_RETRYS parameter specifies the number of times a STATUS message is sent by the MessageStore after applications stop sending messages on a topic. Continued activity on a topic (that is, messages being sent) results in further STATUS messages being sent.

IT_MS_STATUS_TOPIC_LIVE_SECS Default value: 900 seconds (15 minutes)

If the MessageStore does not receive any new messages on a topic for the period of time specified by the IT_MS_STATUS_TOPIC_LIVE_SECS parameter, the topic is deleted. If, and when, traffic on the deleted topic continues, the topic is created and bound once more. This behavior conserves memory on infrequently used topics.

Status messages are sent by the MessageStore for each topic that it considers to be active. A topic becomes inactive if there is a period of inactivity greater than that specified in the IT_MS_STATUS_TOPIC_LIVE_SECS parameter.

IT_ACK_RETRY	Default value: 3

Every message sent on an otsfp topic is received by the MessageStore, stored on a disk-based database and an acknowledgment message sent to the supplier. The IT_ACK_RETRY parameter specifies the number of times a message is re-sent to the MessageStore by a supplier before an exception is raised if an acknowledgment is not received.

IT_ACK_RETRY_TIME	Default value: 11000 (ms)

The IT_ACK_RETRY_TIME parameter specifies the interval between successive re-sends of a message using the otsfp protocol. In combination with the IT_ACK_RETRY parameter, this sets a limit to the time over which the Store and Forward Protocol can handle communication failure without raising an exception.

IT_MAX_ACK_KB	Default value: 1280 Kilobytes

There is a limit to the amount of information that is sent by a supplier before it expects an acknowledgment. The IT_MAX_ACK_KB parameter specifies the maximum size in Kilobytes for the amount of information that can be outstanding before an acknowledgment is received.

IT_MS_COMPACT_INTERVAL	Default value: 3000 ((ms))
IT_MS_COMPACT_BATCH_SIZE	Default value: 300

The otadmin tool is used to compact the MessageStore database. Compaction removes messages that do not need to be kept in the database. These are usually old messages. For more information about the otadmin tool, refer to Chapter 16, "Tools".

The IT_MS_COMPACT_INTERVAL parameter determines the frequency of compaction operations once a compaction is initiated by the otadmin tool. The compaction is performed in batches to improve concurrency/performance of the MessageStore while compaction is in progress. The MessageStore daemon compacts the database by copying the records that need to remain in the database to a new file, then replacing the old database file with the newly created file. The IT_MS_COMPACT_INTERVAL and IT_MS_COMPACT_BATCH_SIZE parameters affect the speed with which the compaction-by-copy operation is performed.

The IT_MS_COMPACT_INTERVAL specifies the period of time between compaction batches, while the IT_MS_COMPACT_BATCH_SIZE parameter specifies the approximate size in Kilobytes that is processed in each batch. It is important that the rate at which compaction is performed is always faster than the rate at which a MessageStore daemon is receiving messages. Otherwise the compaction never completes, as new database entries are added faster than they can be processed for compaction.

In deciding on a suitable rate for compaction (taking into account the fact that the time spent processing entries for compaction is not spent in dealing with incoming messages and the forwarding of those messages), it should be noted that entries that are not copied into the new database are considered to have a processing cost of 128 bytes Entries that are copied are given a minimum processing cost of 3/4 of a Kilobyte, ensuring that the time taken for updating the key values for small entries is taken into account. The compaction algorithm also ensures that at least 3 seconds and no more than 6 seconds is used to process each batch, enabling the messageStore to process incoming messages. it is recommended that compaction is performed during quiet times, for example 2am, to minimize the impact on messages using the Store and Forward Protocol (otsfp).

Using the default settings, the worst case compaction, that is all entries to be copied, takes place at 100 Kb/s. This should be sufficient to ensure that compaction always completes, except in a system where the Store and Forward Protocol (otsfp) is sending more than 100 Kb/s in total.

Directory Enquiries Daemon Configuration Parameters

This section discusses the configuration parameters that affect the performance of the Directory Enquiries daemon (`otd/otdsm`).

`IT_OTD_TOPICSTORE`	Topic Store name. Default Value: `TopicStore`
`IT_OTD_APPLISTORE`	Application Store name. Default Value: `AppliStore`
`IT_OTD_LINKSTORE`	Link Store name. Default value: `LinkStore`

The OrbixTalk Directory Enquiries daemon (`otd`) maintains the following disk-based databases:

- Topic Store

  The Topic Store records the mappings between topic names and IP multicast addresses. The `IT_OTD_TOPICSTORE` parameter specifies the filename used for the log files of the Topic Store. As with the Message Store, two files are created for the Topic Store: one with a ".`dat`" extension, and the other with a ".`ndx`" extension.

- Application Store

  The Application Store records mappings between application names and application IDs. The `T_OTD_APPLISTORE` parameter specifies the filename used for the log files of the Application Store.

- Link Store

  The Link Store records mappings between applications and the topics used by those applications. The `IT_OTD_LINKSTORE` parameter specifies the filename used for the log files of the Link Store.

`IT_OTD_STORE`	Default value: . (current directory)
`IT_APP_STORE`	Default value: . (current directory)
`IT_LINK_STORE`	Default value: . (current directory)

The IT_OTD_STORE parameter specifies the Topic Store directory, the IT_APP_STORE specifies the Application Store directory, and the IT_LINK_STORE specifies the Link Store directory. Ensure the directories already exist and the user ID under which the OrbixTalk Directory Enquiries daemon is executed has permission to read and write to these directories. If the directory does not exist, or does not have the correct access permissions, the OrbixTalk Directory Enquiries daemon reports errors on startup. However, it continues to execute, resulting in later communication failure in the OrbixTalk system when the OrbixTalk Directory Enquiries daemon fails to map Topic Names to multicast addresses correctly. By default, all database files are placed in the directory from which the OrbixTalk Directory Enquiries daemons are started.

IT_DIRENS_NAME	Default value: //OrbixTalk/DirectoryEnquiries

All OrbixTalk applications use a specific topic for communicating with the Directory Enquiries daemon. The IT_DIRENS_NAME configuration parameter specifies the topic used. It must be in a form similar to:

```
//OrbixTalk/DirectoryEnquiries
```

Note: Do not specify a protocol before the opening // of the Topic Name.

IT_DIRENQ_IPADDR	Directory Enquiries IP address. Default Value: 225.0.0.0
IT_DIRENQ_IPADDR_RANGE	Directory Enquiries address range. Default Value: 10 for HPUX10.x; 31 for all other platforms.

Using OrbixTalk, all communication takes place using IP multicast addresses. The range of IP multicast addresses used by an OrbixTalk system is specified by these two parameters. The IT_DIRENQ_IPADDR parameter specifies the first IP address used in the system. It defaults to 225.0.0.0 and must be in a similar form, in the range 225.0.0.0 to 239.255.255.255. The OrbixTalk Directory Enquiries daemon allocates new multicast addresses to topics in an incremental fashion from this address. Many addresses in this range are reserved for specific use by organizations such as the Internet Assigned Numbers Authority (IANA). For the latest list please see the following:

ftp://ftp.isi.edu/in-notes/iana/assignments/multicast-addresses

OrbixTalk is a development tool that can be used for an unlimited range of communications and data types. As such, the onus is on the developer and user to be aware of any issues that can arise as a result of their choice of IP Address range.

The Directory Enquiries address range determines the maximum number of addresses that the OrbixTalk Directory Enquiries daemon can allocate. On some platforms, such as HPUX10.x, there is a hardware/software limitation on the number of multicast addresses that can be allocated.

WARNING: There appears to be a problem with NT if you do all of the following:

---

1. You change the second or third bytes of the base IP address (IT_DIRENQ_IPADDR) to be non-zero (such as 225.0.1.0 or 225.1.0.0)
2. You set IT_DIRENQ_IPADDR_RANGE to be greater than 30.
3. You have more than 32 topics.

If this is the case, then OrbixTalk hangs when detaching from the 33rd and later topics, and the listeners do not receive messages on those topics.

The IT_DIRENQ_IPADDR_RANGE parameter specifies the number of multicast addresses that can be assigned to the system. OrbixTalk multiplexes topics on a single IP address if more topics are created than the range of multicast addresses allows.

Use the IT_DIRENQ_IPADDR_RANGE parameter when you want to run a number of OrbixTalk Directory Enquiries daemons in an environment. When used with IT_DIRENQ_IPADDR, it is possible to guarantee that the multiple OrbixTalk Directory Enquiries daemons never allocate the same (or overlapping) IP addresses.

Although there is no limit on the range that can be specified, all hardware imposes a limit on the number of multicast addresses that can be used on one host. The default value of the range of addresses is the upper limit for the type of host on which the OrbixTalk Directory Enquiries daemon is being used. These values are:

Host	Default value of range of addresses
Windows 95/NT	31
Solaris	31
HP/UX	10

For an OrbixTalk system, you need to ensure that no more than the maximum number of multicast addresses are used on each host. This includes addresses used by programs other than those in the OrbixTalk system. There is a related side-effect of using many multicast addresses on UNIX hosts. Each address used is associated with a number of file descriptors for the ports used on the address. This can easily reach the limit imposed by the operating system on the maximum number of open file descriptors for each process. If the limit is reached, OrbixTalk processes fail to open the network connections required to support communication and subsequently the processes fail. The limit can be modified with the `ulimit -n` command, and should be set high enough so that each process does not run out of file descriptors.

You may need to install different OrbixTalk Directory Enquiries daemons to keep the work of programmers separate from other OrbixTalk environments. To run separate OrbixTalk Directory Enquiries daemons, each environment must have a different value for `IT_DIRENQ_IPADDR` and each `IT_OTD_TOPICSTORE`. The `IT_OTD_STORE` value must be unique.

`IT_DIRENQ_RETRYS`	Default value: 6
`IT_DIRENQ_INTERVAL`	Default value: 5000 (ms)

OrbixTalk applications send requests to the OrbixTalk Directory Enquiries daemon when they start (to obtain an application ID), and when they register objects as a supplier or consumer on a new topic (to obtain the topic ID). Each request has an associated response that is sent by the OrbixTalk Directory Enquiries daemon, but because raw IP multicast is not reliable, it is possible that either the request or response can be lost by the network. All communication with the OrbixTalk Directory Enquiries daemon uses the Directory Enquiries Protocol (DEP), which adds a simple level of reliability.

The IT_DIRENQ_RETRYS parameter specifies the number of times a request is sent to the OrbixTalk Directory Enquiries daemon without a response before an exception is raised. When an OrbixTalk process is initialized, an attempt is made to establish a connection with the OrbixTalk Directory Enquiries daemon. This parameter determines the number of retries made to the OrbixTalk Directory Enquiries daemon before the connect attempt fails.

The IT_DIRENQ_INTERVAL parameter specifies the interval between resends of a request to the OrbixTalk Directory Enquiries daemon before an exception is raised. The Directory Enquiries interval determines the time between attempts to connect to the OrbixTalk Directory Enquiries daemon (see IT_DIRENQ_RETRYS). These are Boot and Application Lookup requests as well as topic bind requests.

The default settings for the IT_DIRENQ_RETRYS and IT_DIRENQ_INTERVAL parameters are sufficient for a normal network. However, a heavily loaded network can require an increase in the values of either or both of these parameters to prevent applications from failing to communicate with the OrbixTalk Directory Enquiries daemon.

IT_DIRENQ_WILD_INTERVAL Default value: 18000000 (ms) (5 hours)

OrbixTalk applications that use wildcard topics send a PROD message to the OrbixTalk Directory Enquiries daemon (otd) at a relatively infrequent interval to inform the OrbixTalk Directory Enquiries daemon that the topic is still in use. Once the OrbixTalk Directory Enquiries daemon no longer hears PROD messages for a particular wildcard topic, the wildcarded topic is removed from the record of wildcard topics that is held in the OrbixTalk Directory Enquiries daemon. The IT_DIRENQ_WILD_INTERVAL parameter specifies the interval between each PROD message. There should be no reason to modify this parameter.

IT_DEFAULT_DIRENS_PORT Default value: 5000

OrbixTalk uses two ports for communications. The IT_DEFAULT_DIRENS_PORT parameter specifies the first of the two port numbers. The second port always uses a port number that is one greater than the first. It is essential that the ports used by an OrbixTalk system are not used by any other process on the same

host. This parameter should be configured to set the port number to a value between 1024 and 65534 where it is known that the two ports are available for use.

Network Configuration Parameters

This section discusses the configuration parameters that affect the performance of the network.

IT_MC_INTERFACE	Default value: 0.0.0.0 (specifies the default network interface)

For machines with more than one network interface, the multicast interface used by OrbixTalk is set using the IT_MC_INTERFACE parameter. When this parameter is set to the IP address of a network interface other than the default interface, OrbixTalk uses that interface for all network communication. So, for example, if you are using a machine with more then one network card, indicate which interface OrbixTalk applications should use. For example:

```
IT_MC_INTERFACE 165.250.232.155
```

A single OrbixTalk application cannot use multiple interfaces.

IT_RECV_SOCKET_BUFF_SIZE	Default value: 65536 (bytes)

The IT_RECV_SOCKET_BUFF_SIZE parameter sets the buffer size used for receiving sockets.

IT_SEND_SOCKET_BUFF_SIZE	Default value: 65536 (bytes)

The IT_SEND_SOCKET_BUFF_SIZE parameter sets the buffer size used for sending sockets.

IT_LIVE_TIME	Default value: 2 (should be in the range 0 to 255). Modify this parameter with caution as changes can affect network security depending on the configuration of network multicast routers.

All UDP packets, including those used for IP multicast, include a time-to-live (TTL) field that determines the extent to which each packet travels through a network. IP routers can be set to decrement the time-to-live (TTL) field of each packet, and forward only those packets with a TTL value that is greater than zero. In this way, a network can be arranged so that multicast packets only reach those points in the network that they are designed to reach. The IT_LIVE_TIME parameter specifies the TTL field for message fragments (packets) multicast on the network. This is dependent on the number of routers required to forward packets onto separate networks.

General Configuration Parameters

This section discusses general configuration parameters.

IT_LOG_LEVEL	Default value: 0 (no output).
IT_LOG_CONSOLE	Console output flag.
	Default Value: 1 (output to console).
IT_LOG_SYSLOG	SYSLOGD output flag.
	Default Value: 0 (no output).
IT_LOG_FLAGS	Logging output settings.
	Default Value: "" (no output).
IT_LOG_TID	Thread ID logging information.
	Default Value: 0 (no thread ID).

Logging information from OrbixTalk applications is useful in determining if there is a problem with OrbixTalk when your application appears to fail.

The IT_LOG_LEVEL parameter specifies the level of logging output, from 0 (no output) to 32. The higher the number the more information is reported (including any previous levels):

0. User.
1. Events.
2. Errors.
3. Warnings.

4. Information.
5. Lower level events.
6. Internal Transport Interface.
7. Message delivery.
8. Store and Forward Protocol.
9. Reliable Multicast Protocol.
10. Directory Enquiries Protocol.
11. Database.
12. Message fragment.
13. Component.
14. Fault Tolerance.
15. Orbix integration.
16. -30. Unused.
32. Timer Events.

The logging output can be sent to the standard output using the IT_LOG_CONSOLE parameter, or to a file using the IT_LOG_SYSLOG parameter. These can be set to 0 (no output) or 1. When the IT_LOG_SYSLOG parameter is set to 1, every OrbixTalk application (including the daemons) logs all output to a file with a name in the following form:

<app name>.<YYMMDD>_<HH.MM.SS>.pid<NNN>.txt

where <app name> is the application name and <NNN> is the process ID. If the application name is of the form a/b/c for example, only the c part is used in the filename. The file is stored in the directory specified by the IT_APP_STORE configuration parameter.

Logging output can also be defined through the IT_LOG_FLAGS configuration parameter. This is a string valued parameter that can be set to a list of logging levels. Available debug logging flags are:

USR, EVT, ERR, WARN, INFO, EVD, ITF, DLV, SFP, EVD, RMP, DEP, DB, FRAG, CPT, IMA, FT, TIM.

For example, IT_LOG_FLAGS can be set to WARN, INFO, DB to add those three logging levels to those already output because of the IT_LOG_LEVEL configuration parameter.

The IT_LOG_TID parameter can be set to 1 to include thread identifier information in the logging output.

IT_OT_DAEMON_BACKGROUND	Start OrbixTalk daemon as a background process. Default Value: 0 (foreground)

Set the IT_OT_DAEMON_BACKGROND parameter to 1 to cause the OrbixTalk daemons to start as background processes on UNIX platforms. This configuration parameter can be overridden using the -F flag on each of the daemons. Similarly, the -B switch on UNIX platforms overrides the parameter to run a daemon as a background process.

Fault Tolerance Configuration Parameters

The following configuration parameters are required for Fault Tolerance. It is recommended that all parameters are set within the orbixtalk3.cfg file, and the same orbixtalk3.cfg file is used for both OrbixTalk daemons; that is, both OrbixTalk daemons have the same IT_CONFIG_PATH parameter setting. This ensures that some parameters are identical for both OrbixTalk daemons in a fault tolerant pair.

IT_OT_FT_ENABLE	Fault Tolerance support. Default Value: 0 (no Fault Tolerance)

Note: IT_OT_FT_ENABLE (default 0) must be set to 1 to enable Fault Tolerance in OrbixTalk daemons. Both OrbixTalk daemons comprising the fault tolerant pair must have this set, otherwise, unexpected behavior can result.

IT_FT_HEART_BEAT_INTERVAL	Default Value: 1000 (ms) Forced minimum of 250 ms.
IT_MAX_FT_HEART_BEAT	Default Value: 6. Forced to minimum of 10 if Fault Tolerance is enabled.

The OrbixTalk Directory Enquiries daemon (`otd`)and OrbixTalk MessageStore daemon (`otmsd`)always start in secondary phase. In the secondary phase, the OrbixTalk daemon does not service requests from other processes, but listens for the heartbeat pings of other OrbixTalk daemons. When the OrbixTalk Directory Enquiries daemon or OrbixTalk MessageStore daemon detects that another `otd` or `otmsd` is running, it stays in secondary phase. If an OrbixTalk daemon in secondary phase does not receive any pings, it moves to the primary phase and begins to service requests.

An OrbixTalk Directory Enquiries daemon or an OrbixTalk MessageStore daemon in the secondary and primary phase sends heartbeat pings at an interval equal to `IT_FT_HEART_BEAT_INTERVAL`. An OrbixTalk Directory Enquiries daemon or an OrbixTalk MessageStore daemon in the secondary phase waits a multiple of `IT_FT_HEART_BEAT_INTERVAL` before going to the primary phase, where the multiple is specified by `IT_MAX_FT_HEART_BEAT`.

The `IT_FT_HEART_BEAT_INTERVAL` parameter determines the frequency at which the OrbixTalk Directory Enquiries daemon and OrbixTalk MessageStore daemon "ping" (in milliseconds). When an OrbixTalk Directory Enquiries daemon or OrbixTalk MessageStore daemon initializes, pings are detected to determine if it should become primary. Currently, running two daemons of the same mode is not allowed; for example, running two Master mode OrbixTalk Directory Enquiries daemons is not allowed—the second daemon fails to start. Running a Master mode OrbixTalk Directory Enquiries daemon and a Slave mode OrbixTalk Directory Enquiries daemon is allowed.

The Daemon Process Detection Tool utility (`otpsd`) also uses these configuration parameters to listen for an OrbixTalk Directory Enquiries daemon or an OrbixTalk MessageStore daemon.

The greater the time taken to move into primary phase, the more tolerant the system becomes to hanging/unresponsive OrbixTalk daemons. The time taken by an OrbixTalk daemon to move into primary phase (primary delay) is considered in the following scenarios:

- Supplier Applications using the Store and Forward Protocol (SFP) —sending messages to the MessageStore daemon (`otmsd`).
- Applications using the Reliable Muliticast Protocol (RMP) or Store and Forward Protocol (SFP)—on start-up (application booting) or binding topics to multicast IP addresses.

Note: The IT_FT_HEART_BEAT_INTERVAL and IT_MAX_FT_HEART_BEAT parameters must be the same for both OrbixTalk daemons in a fault tolerant pair, otherwise fail-over cannot be guaranteed and datastores can be corrupted.

Lock files reside in the same directory as the datastore for each OrbixTalk daemon (otd/otdsm or otmsd). For the Directory Enquiries daemon (otd / otdsm) this directory is set according to the value of the IT_OTD_STORE entry in the orbixtalk3.cfg file or environment variable. The lock file has a fixed name of OTD_FT.lck. The otmsd (MessageStore daemon) lock file resides in the directory specified by the IT_MS_STORE_DIR entry in the orbixtalk3.cfg file or environment variable with the (fixed) name of OTMSD_FT.lck.

Supplier Applications using SFP

IT_ACK_RETRY	Default Value: 3 retries
IT_ACK_RETRY_TIME	Default Value: 5000ms

When an OrbixTalk application sends a message using the Store and Forward protocol, an acknowledgment from the MessageStore daemon (otmsd) is expected within the time (in ms) specified by the IT_ACK_RETRY_TIME configuration parameter. If an acknowledgment is not received, the supplier application re-sends the message for the number of times specified by the IT_ACK_RETRY configuration parameter, at intervals specified by the IT_ACK_RETRY_TIME configuration parameter. The longest time an OrbixTalk MessageStore daemon is unavailable before the supplier application raises an exception is IT_ACK_RETRY_TIME multiplied by IT_ACK_RETRY. This is called the SFP retry period. If the primary delay is equal to N ms, then the SFP retry period must be greater than N. If they are equal and it takes N ms to become primary, there is a chance a message will be rejected.

Applications using RMP or SFP

IT_DIRENQ_RETRYS	Default Value: 6
IT_DIRENQ_INTERVAL	Default Value: 5000 ms

OrbixTalk applications send requests to the Directory Enquiries daemon (otd or otdsm). If a request is not answered in the period of time (in ms) specified by the IT_DIRENQ_INTERVAL configuration parameter, another request is made. This process continues until the number of requests is greater than the number specified by the IT_DIRENQ_RETRYS configuration parameter. The total time a non-primary OrbixTalk Directory Enquiries daemon can be unavailable, before an OrbixTalk application raises an exception, is IT_DIRENQ_INTERVAL multiplied by IT_DIRENQ_RETRYS (OTD retry period). If the primary delay is equal to N ms, the OTD retry period must be greater than N. If they are equal and it takes N ms to become primary, there is a chance an exception will be raised by the OrbixTalk application.

Appendix B
IIOP Gateway Configuration Settings

This appendix details the configuration file variables used by the OrbixTalk IIOP Gateway.The Gateway configuration variables are contained in the scope OrbixTalk.Gateway. You can adjust these settings with the Orbix configuration tool.

Variable	Effect
`IT_EVENTS_NOT_ORBIX_SERVER`	When this variable is set to `YES`, then OrbixTalk does not call `impl_is_ready()`. Default is `NO`.. For example: `IT_EVENTS_NOT_ORBIX_SERVER = "NO";`
`IT_EVENTS_SERVER_NAME`	Server name used in call to `impl_is_ready()`. Default is `ES`. For example: `IT_EVENTS_SERVER_NAME = "ES";`
`IT_DEFAULT_TX_TIMEOUT`	Timeout value in milliseconds passed to `defaultTxTimeout()`. Default is infinite. For example: `IT_DEFAULT_TX_TIMEOUT = 60000;`
`IT_SERVER_TIMEOUT`	Timeout value in milliseconds passed to `processEvents()`. Default is infinite. `IT_SERVER_TIMEOUT = 60000;`

Table B.1: *Gateway Configuration Variables*

Variable	Effect
IT_SET_DIAGNOSTICS	Value passed to setDiagnostics(). Default is 1. Valid values are 0, 1 and 2. For example: IT_SET_DIAGNOSTICS = "0";
IT_USE_TRANSIENT_PORT	When this variable is set to YES, OrbixTalk calls useTransientPort(1). Default is NO. For example: IT_USE_TRANSIENT_PORT = "NO";
IT_WRITE_IOR	When this variable is set to YE", OrbixEventsAdmin IOR is written to the file OrbixEventsAdmin.ref. Default is NO. For example: IT_WRITE_IOR = "NO";
IT_EVENTS_PULL_PROD_TYPE	When a PullConsumer executes a pull() on the ProxyPullSupplier provided by the event server, this variable determines whether to attempt pull() or try_pull() on any PullSuppliers connected. Default is PULL, and the alternative is TRY_PULL. For example: IT_EVENTS_PULL_PROD_TYPE = "PULL";
IT_EVENTS_PULL_PROD_INTERVAL	This value sets the interval in milliseconds between each attempted try-pull() or pull() on connected PullSuppliers. Default is 1000. For example: IT_EVENTS_PULL_PROD_INTERVAL = "1000";
IT_EVENTS_TRY_PULL_DURATION	Determines how long in milliseconds a PullConsumer waits for an event after executing a try_pull() on the ProxyPullSupplier provided by the event server. Default is 100. For example: IT_EVENTS_TRY_PULL_DURATION = "100";

Table B.1: *Gateway Configuration Variables*

Variable	Effect
IT_ROBUST_EVENT_CHANNELS	When this variable is set to YES, event channels are not destroyed by calls to destroy() if Proxies exist. Default is NO. For example: IT_ROBUST_EVENT_CHANNELS = "NO";
IT_INITIAL_UNTYPED_EVENT_CHANNELS	This variable causes OrbixTalk to create untyped events channels created at start-up with the channel name provided. Default is "". For example: IT_INITIAL_UNTYPED_EVENT_CHANNELS = test_channel;
IT_INITIAL_TYPED_EVENT_CHANNELS	This variable causes OrbixTalk to create typed events channels created at start-up with the channel name provided. Default is "". For example: IT_INITIAL_TYPED_EVENT_CHANNELS = test_channel;

Table B.1: *Gateway Configuration Variables*

Appendix C
CORBA Event Service: IDL Interfaces

This appendix lists the IDL interfaces for the CORBA Event Service.

The CosEventComm Module

```
// IDL
module CosEventComm {
   exception Disconnected {
   };

   interface PushConsumer {
      void push (in any data) raises (Disconnected);
      void disconnect_push_consumer ();
   };

   interface PushSupplier {
      void disconnect_push_supplier( );
   };
   interface PullSupplier {
      any pull () raises (Disconnected);
      any try_pull (out boolean has_event) raises (Disconnected);
      void disconnect_pull_supplier();
   };
```

```
   interface PullConsumer {
      void disconnect_pull_consumer ();
   };
};
```

The CosEventChannelAdmin Module

```
// IDL
module CosEventChannelAdmin {

   exception AlreadyConnected {
   };

   exception TypeError {
   };

   interface ProxyPushConsumer : CosEventComm::PushConsumer {
      void connect_push_supplier (
         in CosEventComm::PushSupplier push_supplier)
         raises (AlreadyConnected);
   };
   interface ProxyPullSupplier : CosEventComm::PullSupplier {
      void connect_pull_consumer (
         in CosEventComm::PullConsumer pull_consumer)
         raises (AlreadyConnected);
   };

   interface ProxyPullConsumer : CosEventComm::PullConsumer {
      void connect_pull_supplier (
         in CosEventComm::PushSupplier pull_supplier)
         raises (AlreadyConnected, TypeError);
   };

   interface ProxyPushSupplier : CosEventComm::PushSupplier {
      void connect_push_consumer (
         in CosEventComm::PushConsumer push_consumer)
         raises (AlreadyConnected, TypeError);
   };
```

```
   interface ConsumerAdmin {
      ProxyPushSupplier obtain_push_supplier ();
      ProxyPullSupplier obtain_pull_supplier ();
   };

   interface SupplierAdmin {
      ProxyPushConsumer obtain_push_consumer ();
      ProxyPullConsumer obtain_pull_consumer ();
   };

   interface EventChannel {
      ConsumerAdmin for_consumers ();
      SupplierAdmin for_suppliers ();
      void destroy ();
   };
};
```

The CosTypedEventComm Module

```
// IDL
module CosTypedEventComm {

   interface TypedPushConsumer : CosEventComm::PushConsumer {
      Object get_typed_consumer ();
   };

   interface TypedPullSupplier : CosEventComm::PullSupplier {
      Object get_typed_supplier ();
   };
};
```

The CosTypedEventChannelAdmin Module

```
module CosTypedEventChannelAdmin {

   exception InterfaceNotSupported {
   };

   exception NoSuchImplementation {
   };

   typedef string Key;

   interface TypedProxyPushConsumer :
         CosEventChannelAdmin::ProxyPushConsumer,
         CosTypedEvemtComm::TypedPushConsumer {
   };

   interface TypedProxyPullSupplier :
         CosEventChannelAdmin::ProxyPullSupplier,
         CosTypedEventComm::TypedPullSupplier {
   };

   interface TypedSupplierAdmin :
         CosEventChannelAdmin::SupplierAdmin {
      TypedProxyPushConsumer obtain_typed_push_consumer (
               in Key supported_interface)
               raises (InterfaceNotSupported);
      ProxyPullConsumer obtain_typed_pull_consumer (
               in Key uses_interface)
               raises (NoSuchImplementation);
   };

   interface TypedConsumerAdmin :
            CosEventChannelAdmin::ConsumerAdmin {
      TypedProxyPullSupplier obtain_typed_pull_supplier (
               in Key supported_interface)
               raises (InterfaceNotSupported);
      ProxyPushSupplier obtain_typed_push_supplier (
               in Key uses_interface)
               raises (NoSuchImplementation);
   };
```

```
   interface TypedEventChannel {
      TypedConsumerAdmin for_consumers ();
      TypedSupplierAdmin for_suppliers ();
      void destroy ();
   };
};
```

The OrbixTalkAdmin Module

```
module OrbixTalkAdmin
{
  typedef string                ChannelName;

  exception InvalidName{ };   // The channel/application name
                              // supplied is illegal
  exception InvalidOption{ }; // Invalid Replay Type/Value
                              // combination
  exception AlreadySet{ };    // app name already set - SFP

  // Replay Type
  //
  // REPLAY_NONE     - Do not replay messages - Default
  // REPLAY_ALL      - Replay all messages
  // REPLAY_RELATIVE - Replay n messages relative to most recent
  // REPLAY_ABSOLUTE - Replay messages starting from n
   // REPLAY_USE_EXISTING   - Use the existing replay mechanism for
the channel
  //
  typedef unsigned short ReplayType;
  typedef unsigned long  ReplayValue;

  const ReplayType REPLAY_NONE         = 0;
  const ReplayType REPLAY_ALL          = REPLAY_NONE + 1;
  const ReplayType REPLAY_RELATIVE     = REPLAY_NONE + 2;
  const ReplayType REPLAY_ABSOLUTE     = REPLAY_NONE + 3;
   const ReplayType REPLAY_USE_EXISTING = REPLAY_NONE + 6;

  interface OTChannelManager
  {
    CosEventChannelAdmin::EventChannel get_event_channel
    (
```

```
      in    ChannelName  channel_name,
      inout ReplayType   type,
      inout ReplayValue  value
    )       raises (InvalidName,InvalidOption);

    CosTypedEventChannelAdmin::TypedEventChannel
get_typed_event_channel
    (
      in    ChannelName  channel_name,
      inout ReplayType   type,
      inout ReplayValue  value
    )       raises (InvalidName,InvalidOption);

    // required for SFP
    //
    void set_app_name
    (
      in string name
    ) raises (InvalidName, AlreadySet);
  };
};
```

Appendix D
Using the OrbixTalk API Directly

OrbixTalk suppliers and consumers are normally implemented with the event service. However, you can also write applications using the OrbixTalk API.

In previous versions of OrbixTalk, you could develop multicast applications with the OrbixTalk API. It is now simpler and easier to use the interfaces provided by the Event Service as described in Part II Developing OrbixTalk Applications, "Part II Developing OrbixTalk Applications". However, the OrbixTalk API is still available for you to use.

In discussing the OrbixTalk API, *suppliers* are referred to as *talkers*, and *consumers* are called *listeners*.

This appendix demonstrates how to use the OrbixTalk API. A simple financial application is developed where talkers quote stock prices and listeners listen for stock price quotes. This appendix also discusses how to use the Message Store, and describes the OrbixTalk API demonstration program, `OTAuction`.

A knowledge of basic Orbix programming, as explained in the *Orbix Programmer's Guide*, is assumed. Appendix E, "OrbixTalk Class Reference" provides a reference to the OrbixTalk API.

Overview

There are three stages to developing an application using the OrbixTalk Application Programming Interface (API) directly:

1. Create an IDL interface. The IDL interface is compiled to check the specification and map the IDL interface into C++. The IDL interface is implemented in a C++ class as explained in the *Orbix Programmer's Guide*.

2. Write a listener application that creates objects, registers these as listeners, and awaits messages relevant to the objects' Topic Names.
3. Write a talker application that creates proxy objects and registers these as talkers; operation invocations on these proxies are multicast to any listeners listening on the talkers' Topic Names.

Creating an IDL Interface

To develop a simple financial application where talkers quote stock prices and listeners listen for stock price quotes, the relevant IDL interface is:

```
//IDL interface
interface stockPrice
{
  oneway void quote (in float f);
};
```

A version of this example is available in your OrbixTalk installation.

The `quote()` operation must be defined as IDL `oneway` since talkers can invoke only oneway operations because talkers and listeners are decoupled.

Implement interface StockPrice using the C++ class `StockPrice_i`. In this example, the implementation of the `quote()` operation notifies changes to stock prices as shown below:

```
#include "Server.h"
#include <iostream.h>
void StockPrice_i::quote
(
  float               price,
  CORBA(Environment)&
)
{
  cout << "Stock: " << _marker () << " now at " << price << endl;
}
```

Creating a Listener Application

The listener application creates `StockPrice` objects, registers them as listeners, and waits to receive messages. The code is:

```
// C++
// OrbixTalk Listener.

#include "stock.hh"
#include "Server.h"
#include <iostream.h>

#include <stdlib.h>
#include <orbixTalk.h>

int main()
{
  cout << endl << "OrbixTalk API Stock Price listener :";
  cout << "(Uses the OrbixTalk RMP protocol)" << endl;
  cout << endl << endl;


  OrbixTalk* otalk;
  stockPrice_ptr sunStk;
  stockPrice_ptr ibmStk;
  stockPrice_ptr ionaStk;

  try
  {
    // Initialise OrbixTalk
    //
    otalk = OrbixTalk::initialise();

    // Build various Listener objects
    //
    sunStk = new StockPrice_i("sun");     // "sun" == "otrmp//sun"
    ibmStk = new StockPrice_i("ibm");
    ionaStk = new StockPrice_i("iona");

    // Register the listeners
    //
    otalk->registerListener(sunStk);
    otalk->registerListener(ibmStk);
```

```
    otalk->registerListener(ionaStk);

    // Need to execute the event loop
    // to process incoming events.
    //
    CORBA(Orbix).processEvents(60 * 1000);

    otalk->unregister(ibmStk);
    otalk->unregister(sunStk);
    otalk->unregister(ionaStk);

    CORBA(release)(ibmStk);
    CORBA(release)(sunStk);
    CORBA(release)(ionaStk);
  }
  catch (CORBA(SystemException) &sysEx)
  {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    if (otalk)
    {
      otalk->terminate(1);
    }
    exit(1);
  }
  catch (...)
  {
    cerr << "Unexpected exception " << endl;
    if (otalk)
    {
      otalk->terminate(1);
    }
    exit(1);
  }
  if (otalk)
  {
    otalk->terminate();
  }
  cout << "--- Listener end..." << endl;
  return 0;
}
```

The first step is to initialize OrbixTalk and obtain a reference to it by calling the function `OrbixTalk::initialise()`.

The code then creates three objects. For each object, a marker is specified in the constructor's parameter. This marker is in the form of an OrbixTalk Topic Name:

`<protocol>//<topic identifier>`

where `<protocol>` is one of the following:

```
otmcp (OrbixTalk Raw Multicast Protocol)
otrmp (OrbixTalk Reliable Multicast Protocol)
otsfp (OrbixTalk Store and Forward Protocol)
```

In the example code, the protocol is not specified as the default OrbixTalk Reliable Multicast Protocol (`otrmp`) is used:

```
sunStock = new StockPrice_i("sun");
```

If the marker is in the form of an OrbixTalk Topic Name, this topic is used in the subsequent call to `OrbixTalk::registerListener()`. If the application needs two different objects registered as listeners on the same topic, the constructor's parameter must specify both a marker and a server, and the server is used for the topic. For example:

```
ionaStk = new StockPrice_i("Iona_SP:otrmp//iona");
```

When both a marker and server are specified in the constructor, you must qualify the protocol used in the Topic Name; `"Iona_SP:iona"` would be incorrect.

For each object, the call to `OrbixTalk::registerListener()` registers the object as a listener. The call also contacts the OrbixTalk Directory Enquiries daemon to obtain a multicast address corresponding to the topic that the listener is listening on. At this point, there are three listener objects listening on the topics — `sun`, `ibm` and `iona`.

The call to `CORBA::Orbix.processEvents()` indicates the application's readiness to accept Orbix events — in this case, incoming messages. If this application is also to act as a server for normal (non-OrbixTalk) invocations on `StockPrice` or other objects, `impl_is_ready()` should be called (with a zero timeout). Because `impl_is_ready()` has not been called here, the Orbix daemon has no knowledge of this server. Instead, the Directory Enquiries

daemon uses the mapping of Topic Names to IP multicast addresses to set up the initial connection. Subsequent invocations are directed via multicast to the appropriate object(s).

Creating a Talker Application

The talker application creates three `StockPrice` proxy objects and registers them as talkers. The `StockPrice` proxy objects then send messages, quoting stock prices.

```
// C++
// Talker Application.

#include "stock.hh"
#include <iostream.h>
#include <stdlib.h>

#include <orbixTalk.h>

OrbixTalk_ptr otalk = 0;

// utility routine to register a Talker
// Note: This should be called from within a try/catch block.
//
stockPrice_ptr myRegisterTalker(const char *stk)
{
  stockPrice_ptr pResult;

  // Get a CORBA::Object proxy built for us
  //
  CORBA(Object_ptr) obj = otalk->registerTalker
                          (
                            stk,
                            stockPrice_IR
                          );

  // Narrow it into a stockPrice (the narrow performs an implicit
  // duplicate on obj).
  //
  pResult = stockPrice::_narrow(obj);
  CORBA::release(obj);
```

```
  return pResult;
}

// This timer gets plugged into the OrbixTalk message loop.
//
class stockTimer : public OrbixTalk::TimerEvent
{
  char*          m_stock_name;
  unsigned long  m_last_price;
  stockPrice_ptr m_stock;

public:
  stockTimer
  (
    char*          stock_name,
    unsigned int   t,
    stockPrice_ptr stock,
    unsigned long  start_price
  ) :
    OrbixTalk::TimerEvent(t)
  {
    m_stock_name = new char[strlen(stock_name) + 1];
    strcpy(m_stock_name, stock_name);
    m_stock = stock;
    m_last_price = start_price;
  }

  ~stockTimer()
  {
    delete [] m_stock_name;
  }

  void fired()
  {
    try
    {
      cout << "Quoting " << m_stock_name << " @ " << m_last_price
           << endl;
      m_stock->quote(m_last_price);
      m_last_price = m_last_price + 10;
      cout << "Done quote on " << m_stock_name << endl;
```

```
      // Prime the timer again if we want to quote again
      //
      if (m_last_price < 10000)
      {
        otalk->addTimerEvent(this);
      }
    }
    catch(CORBA(SystemException)& sysEx)
    {
      cerr << "Unexpected system exception" << endl;
      cerr << &sysEx;
      otalk->terminate(1);
      exit(1);
    }
    catch(...)
    {
      cerr << "Unexpected exception" << endl;
      otalk->terminate(1);
      exit(1);
    }
  }
};

int main()
{
  int result = 0;

  cout << endl << "OrbixTalk API Stock Price talker :";
  cout << "(Uses the OrbixTalk RMP protocol)" << endl;
  cout << endl << endl;

  try
  {
    // Initialise OrbixTalk
    //
    otalk = OrbixTalk::initialise();

    stockPrice_ptr sunStk;
    stockPrice_ptr ibmStk;
    stockPrice_ptr ionaStk;

    // Register the talker objects
    //
```

```
    sunStk = myRegisterTalker("sun");  // "sun" == "otrmp//sun"
    ibmStk = myRegisterTalker("ibm");
    ionaStk = myRegisterTalker("iona");

    stockTimer* sunTimer = new stockTimer("Sun", 150, sunStk, 10);
    stockTimer* ibmTimer = new stockTimer("IBM", 300, ibmStk, 20);
    stockTimer* ionaTimer = new stockTimer
                                (
                                  "Iona", 100,
                                  ionaStk, 5
                                );

    otalk->addTimerEvent(sunTimer);
    otalk->addTimerEvent(ibmTimer);
    otalk->addTimerEvent(ionaTimer);

    // Wait for a minute
    //
    CORBA(Orbix).processEvents(60 * 1000);

    // Delete the timers.  This will remove them from
    // the OrbixTalk timer service if they are in it.
    //
    delete sunTimer;
    delete ibmTimer;
    delete ionaTimer;

    // Unregister the talker objects
    //
    otalk->unregister(ionaStk);
    otalk->unregister(ibmStk);
    otalk->unregister(sunStk);
  }
  catch(CORBA(SystemException) & sysEx)
  {
    cerr << "Unexpected system exception, exiting" << endl;
    cerr << &sysEx;
    result = 1;
  }
  catch(...)
  {
    cerr << "Unexpected exception " << endl;
    result = 1;
```

```
  }

  // Terminate OrbixTalk
  //
  if (otalk)
  {
    otalk->terminate(result);
  }

  cout << endl << "Stock Talker end..." << endl << endl;

  return result;
}
```

Initialize OrbixTalk by calling the function `OrbixTalk::initialise()`.

Note: `OrbixTalk::initialise()` must be called before any other Orbix or OrbixTalk API call or remote invocation.

The code then declares three proxy objects and registers them as talkers on the topics `sun`, `ibm` and `iona`. As before, these talkers use the default OrbixTalk Reliable Multicast Protocol (`otrmp`). The `registerTalker()` function returns an Orbix proxy object that must be narrowed to an object of the desired type. `OrbixTalk::registerTalker()` is declared as:

```
// C++
//
virtual CORBA(Object_ptr) registerTalker
(
  const char*         pServerName,
  const char*         pTypeName,
  CORBA(Environment)& rEnv        = CORBA(default_environment)
);
```

This version of `OrbixTalk::registerTalker()` is similar to Orbix `_bind()` as it accepts a marker:server pair and interface name and returns a proxy object. The `pServerName` parameter of this function specifies the Topic Name on which the talker will send messages. It is in the form of a marker:server pair, where either the marker or server part is optional. If a server name is specified, it must be in the form of a valid OrbixTalk Topic Name, with a protocol specified.

A version of `registerTalker()` that enables an existing object (proxy) to be registered as an OrbixTalk talker is also provided in class `OrbixTalk`. This might be used in conjunction with the Orbix Naming Service because it assumes that a proxy has already been created.

The `registerTalker()` operations contact the Directory Enquiries daemon to register the Topic Name and map it to an IP multicast address.

Once a proxy is registered as an OrbixTalk talker, *only* oneway operations can be called *on that proxy*. An attempt to invoke a normal two-way operation raises a `CORBA::COMM_FAILURE` exception. In the example, the oneway operation `quote()` is invoked on the talker proxies.

OrbixTalk Events

The OrbixTalk Reliable Multicast Protocol (`otrmp`) uses multiple threads within the OrbixTalk library. These threads are created on the call to `OrbixTalk::initialise()`, and terminated on the call to `OrbixTalk::terminate()`.

On the talker side, these threads:

- Handle requests to resend message fragments.
- Time out old messages.
- Periodically send information messages.

On the listener side, these threads:

- Handle incoming messages.

To dispatch incoming messages so that application code is invoked when an OrbixTalk message arrives at a listener, the Orbix event loop must be processing events. Just as normal Orbix server applications initialize and enter the Orbix event loop by calling `CORBA::Orbix.impl_is_ready()` or `CORBA::Orbix.processEvents()`, OrbixTalk listener applications must enter the same event loop.

In the same manner as Orbix, OrbixTalk applications can process the event loop in a number of ways. Normally, listeners perform initialization steps and then enter the Orbix event loop using `CORBA:Orbix.processEvents()`. If an application needs to take greater control of the Orbix event loop, it can be

written so that it enters the event loop for a period of time to process pending events before continuing with other work. An application can determine if there are events pending using `CORBA:BOA:isEventPending()`.

An application can determine when the internal OrbixTalk threads are idle using `OrbixTalk::isIdle()`. However, there are very few instances where an application needs information on the internal processing of OrbixTalk.

On termination of an application, it is important that the OrbixTalk threads are idle so that it is known that all messages have been sent correctly. The `OrbixTalk::terminate()` function can be used to do this. For more information about `OrbixTalk::terminate()`, refer to "OrbixTalk Class" on page 219.

OrbixTalk Timer Events

This section describes how to insert a timed callback into the OrbixTalk timer service. OrbixTalk provides the class `OrbixTalk::TimerEvent`, which is an abstract base class that defines the interface for user timer events.

To create a user timer event, specify a subclass of the abstract base class `OrbixTalk::TimerEvent` which defines the interface for user timer events. You can then add one or more instances of the subclass to the OrbixTalk user timer events loop.

In the following example, class `StockPriceTimer` implements a timer for the `StockPrice` application:

```
// C++
#include <orbixTalk.h>
...

class StockPriceTimer : public OrbixTalk::TimerEvent
{
  char* m_stock_name;
  unsigned long m_last_price;
  StockPrice_ptr m_stock;

public:
  StockPriceTimer
  (
    char* stock_name,
```

```
    unsigned int t,
    StockPrice_ptr stock,
    unsigned long start_price
  ) :
    OrbixTalk::TimerEvent(t)
  {
    m_stock_name = new char[strlen(stock_name) + 1];
    strcpy(m_stock_name,stock_name);
    m_stock = stock;
    m_last_price = start_price;
  }

  ~stockTimer()
  {
    delete[] m_stock_name;
  }

  void fired()
  {
    try
    {
      cout << "Quoting "
           << m_stock_name << " @ "
           << m_last_price << endl;

      m_stock->quote(m_last_price);
      m_last_price = m_last_price + 10;
      cout << "Done quote on "
           << m_stock_name << endl;
    } catch ... // Handle exceptions.

    // Prime the timer again:
    otalk->addTimerEvent(this);
  }
};
```

The constructor of class `CORBA::TimerEvent` takes one parameter specifying, in milliseconds, the timeout for the event. The function `fired()` is called on the user timer event when the timeout expires. In this example, `fired()` is implemented so that it sends messages by invoking the `quote()` operation on a StockPrice object, then re-inserts the user timer event into the OrbixTalk timer service by calling `OrbixTalk::addTimerEvent()`.

```
// C++
// Talker application.
...
OrbixTalk* otalk;
...
int main(int argc, char** argv)
{
  ...
  StockPrice_var ionaStk;
  StockPriceTimer* ionaTimer;
  try
  {
    otalk = OrbixTalk::initialise();

    // Register the talkers.
    ionaStk = myRegisterTalker("otrmp//iona");
    ionaTimer = new StockPriceTimer("IONA", 150, ionaStk, 10);
    otalk->addTimerEvent(ionaTimer);

    // Enter OrbixTalk event loop for one
    // minute, then exit the application.
    CORBA::Orbix.processEvents (60*1000);
    delete ionaTimer;
  } catch ... // Handle exceptions here.

  otalk->terminate();

  return 0;
}
```

The talker application creates an instance of `StockPriceTimer` and inserts it into the user timer events loop by calling `addTimerEvent()` on OrbixTalk. The application enters the OrbixTalk user timer events loop by calling `CORBA::Orbix.processEvents()`.

Using MessageStore with the OrbixTalk API directly

This section modifies the example in Appendix D "Using the OrbixTalk API Directly" on page 219 to use the Store and Forward Protocol provided by the OrbixTalk MessageStore.

The changes required to the example are minimal and are shown in bold text.

Creating a Listener Application

A listener application that uses the OrbixTalk MessageStore is coded as follows:

```
// C++
// OrbixTalk Listener.

#include "stock.hh"
#include "Server.h"
#include <iostream.h>

#include <stdlib.h>
#include <orbixTalk.h>

int main()
{
  char* appName = "//stocksfp/listener";

  cout << endl << "OrbixTalk API Stock Price listener :";
  cout << "(Uses the OrbixTalk SFP protocol)" << endl;
  cout << endl << endl;

  OrbixTalk* otalk;
  stockPrice_ptr sunStk;
  stockPrice_ptr ibmStk;
  stockPrice_ptr ionaStk;

  try
  {
    // Initialise OrbixTalk
    //
    otalk = OrbixTalk::initialise();

    // Set the persistent application name
```

```
    //
    otalk->setPersistentAppName(appName);

    // Build various Listener objects
    //
    sunStk = new StockPrice_i("otsfp//sun");
    ibmStk = new StockPrice_i("otsfp//ibm");
    ionaStk = new StockPrice_i("otsfp//iona");

    // Register the listeners
    //
    otalk->registerListener(sunStk, OrbixTalk::REPLAY_ALL);
    otalk->registerListener(ibmStk, OrbixTalk::REPLAY_ALL);
    otalk->registerListener(ionaStk, OrbixTalk::REPLAY_ALL);

    // Need to execute the event loop
    // to process incoming events.
    //
    CORBA(Orbix).processEvents(60 * 1000);

    otalk->unregister(ibmStk);
    otalk->unregister(sunStk);
    otalk->unregister(ionaStk);

    CORBA(release)(ibmStk);
    CORBA(release)(sunStk);
    CORBA(release)(ionaStk);
  }
  catch (CORBA(SystemException) &sysEx)
  {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    if (otalk)
    {
      otalk->terminate(1);
    }
    exit(1);
  }
  catch (...)
  {
    cerr << "Unexpected exception " << endl;
    if (otalk)
    {
```

```
      otalk->terminate(1);
    }
    exit(1);
  }
  if (otalk)
  {
    otalk->terminate();
  }

  cout << "--- Listener end..." << endl;

  return 0;
}
```

The listener must specify a unique and persistent application name by calling `OrbixTalk::setPersistentAppName()`. For more information about persistent application names, refer to "Persistent Application Name" on page 14. Each talker or listener using the OrbixTalk Store and Forward Protocol maintains a persistent state. The application name is used to find the location of the persistent state.

The listener specifies the protocol using the prefix `otsfp` (Store and Forward Protocol). By default, all stored messages are replayed, however you can choose the type of replay used for the listener. The second parameter of `OrbixTalk::registerListener()` specifies the replay type used for a topic using the Store and Forward Protocol (`otsfp`). The value is an `OrbixTalk::REPLAY_TYPE` and is one of the following:

`REPLAY_NONE = 0`	No replay.
`REPLAY_ALL = 1`	Replay all messages not yet heard.
`REPLAY_LAST = 2`	If messages are missed, replay the most recent message.

Creating a Talker Application

In a talker application, the application name identifies the talker's persistent state. The application name is set using `OrbixTalk::setPersistentAppName()`. Persistent application names must be unique within an OrbixTalk system.

An invocation on an OrbixTalk object does not return until the OrbixTalk MessageStore saves the message to disk and the talker has received an acknowledgment from the OrbixTalk MessageStore.

Note: This example does not use timers in the event loop.

```
// C++
// Talker Application.

#include "stock.hh"
#include <iostream.h>

#include <stdlib.h>
#include <orbixTalk.h>

OrbixTalk_ptr otalk = 0;

// utility routine to register a Talker
// Note: This should be called from within a try/catch block.
//
stockPrice_ptr myRegisterTalker(const char *stk)
{
  stockPrice_ptr pResult;

  // Get a CORBA::Object proxy built for us
  //
  CORBA(Object_ptr) obj = otalk->registerTalker
                          (
                            stk,
                            stockPrice_IR
                          );

  // Narrow it into a stockPrice (the narrow performs an implicit
  // duplicate on obj).
  //
  pResult = stockPrice::_narrow(obj);
```

```
  CORBA::release(obj);

  return pResult;
}

// This timer gets plugged into the OrbixTalk message loop.
//
class stockTimer : public OrbixTalk::TimerEvent
{
  char*          m_stock_name;
  unsigned long  m_last_price;
  stockPrice_ptr m_stock;

public:
  stockTimer
  (
    char*          stock_name,
    unsigned int   t,
    stockPrice_ptr stock,
    unsigned long  start_price
  ) :
    OrbixTalk::TimerEvent(t)
  {
    m_stock_name = new char[strlen(stock_name) + 1];
    strcpy(m_stock_name, stock_name);
    m_stock = stock;
    m_last_price = start_price;
  }

  ~stockTimer()
  {
    delete [] m_stock_name;
  }

  void fired()
  {
    try
    {
      cout << "Quoting " << m_stock_name << " @ " << m_last_price
           << endl;
      m_stock->quote(m_last_price);
      m_last_price = m_last_price + 10;
      cout << "Done quote on " << m_stock_name << endl;
```

```
      // Prime the timer again if we want to quote again
      //
      if (m_last_price < 1000)
      {
        otalk->addTimerEvent(this);
      }
    }
    catch(CORBA(SystemException)& sysEx)
    {
      cerr << "Unexpected system exception" << endl;
      cerr << &sysEx;
      otalk->terminate(1);
      exit(1);
    }
    catch(...)
    {
      cerr << "Unexpected exception" << endl;
      otalk->terminate(1);
      exit(1);
    }
  }
};

int main()
{
  char* appName = "//stocksfp/talker1";
  int   result  = 0;

  cout << endl << "OrbixTalk API Stock Price talker :";
  cout << "(Uses the OrbixTalk SFP protocol)" << endl;
  cout << endl << endl;
  try
  {
    // Initialise OrbixTalk
    //
    otalk = OrbixTalk::initialise();

    // Set the persistent application name
    //
    otalk->setPersistentAppName(appName);

    stockPrice_ptr sunStk;
```

```
    stockPrice_ptr ibmStk;
    stockPrice_ptr ionaStk;

    // Register the talker objects
    //
    sunStk = myRegisterTalker("otsfp//sun");
    ibmStk = myRegisterTalker("otsfp//ibm");
    ionaStk = myRegisterTalker("otsfp//iona");

    stockTimer* sunTimer = new stockTimer("Sun", 100, sunStk, 10);
    stockTimer* ibmTimer = new stockTimer("IBM", 200, ibmStk, 20);
    stockTimer* ionaTimer = new stockTimer
                                (
                                  "Iona", 300,
                                  ionaStk, 5
                                );

    otalk->addTimerEvent(sunTimer);
    otalk->addTimerEvent(ibmTimer);
    otalk->addTimerEvent(ionaTimer);

    // Wait for a minute
    //
    CORBA(Orbix).processEvents(60 * 1000);

    // Delete the timers.  This will remove them from
    // the OrbixTalk timer service if they are in it.
    //
    delete sunTimer;
    delete ibmTimer;
    delete ionaTimer;

    // Unregister the talker objects
    //
    otalk->unregister(ionaStk);
    otalk->unregister(ibmStk);
    otalk->unregister(sunStk);
}
catch(CORBA(SystemException) & sysEx)
{
    cerr << "Unexpected system exception, exiting" << endl;
    cerr << &sysEx;
    result = 1;
```

```
  }
  catch(...)
  {
    cerr << "Unexpected exception " << endl;
    result = 1;
  }

  // Terminate OrbixTalk
  //
  if (otalk)
  {
    otalk->terminate(result);
  }

  cout << endl << "Stock Talker end..." << endl << endl;

  return result;
}
```

OrbixTalk Demonstration Program

The `OTAuction` demonstration program shows how the facilities provided by OrbixTalk can simplify the way in which a distributed application is designed and developed. It is assumed that the reader is familiar with basic Orbix and OrbixTalk programming.

Overview

The `OTAuction` system provides an example of a real-world auction of an item such as a painting. In general, there is a single auctioneer that has information about the painting, and where the auction will be held. The auctioneer has no knowledge of the bidders involved in the auction until the time at which the auction takes place. There is also a group of bidders who have seen the information about the painting, and know where the auction will be held. The auctioneer opens the auction, bidders make bids, the auctioneer informs the group of bidders of the current bidding value as the auction progresses, and eventually the auctioneer closes the auction, either selling the painting, or passing it in.

This scenario is well-suited to implementation using OrbixTalk because all the components involved are distinct and decoupled. Communication takes place asynchronously, with bidders making bids and the auctioneer responding to the group of bidders as a whole.

The `OTAuction` system can also be applied readily to other situations and problems as a design pattern. For example, a set of load-balancing servers can bid for the right to perform an action, with the bidding power of each server being inversely proportional to its current load. An auction is an efficient way of determining how to assign a resource when the objects to which the assignment can be made may not be known in advance.

The `OTAuction` system consists of two programs:

1. Auctioneer program

   The auctioneer program describes the items that are available for auction and uses a number of `Auctioneer_i` objects to coordinate the bidding for the items. The system can be run with multiple `Auctioneer_i` objects.

2. Bidder program

   The bidder program uses a number of `Bidder_i` objects to make bids on the items. The system can be run with multiple `Bidder_i` objects.

These programs are described in more detail in "OTAuction Components" on page 247.

The `OTAuction` system also demonstrates the following advanced features of OrbixTalk programming:

- Combined use of the Topic Names using the Store and Forward Protocol (`otsfp`) and Topic Names using the Reliable Multicast Protocol (`otrmp`) for talking and listening in one process.
- Management of persistent information using the OrbixTalk MessageStore.
- Using dynamic OrbixTalk Topic Names.
- Analogy between OrbixTalk topics and Orbix stringified object references.
- Using wildcarded OrbixTalk topics.
- Mapping derived or polymorphic IDL interfaces to the OrbixTalk Topic Name.

OTAuction System

The basic requirement for `OTAuction` is to enable a number of processes to bid for a set of items. The items to be auctioned are defined at runtime, and the `Auctioneer_i` objects responsible for coordinating the auction of the items have no knowledge of the `Bidder_i` objects that can make bids.

Creating the IDL Interfaces

The bidding process is defined in terms of the following IDL interfaces in the `Auction.idl` file:

- Auction interface
- Auctioneer interface
- Bidder interface
- Observer interface

The Auction Interface

`Auction.idl`

```
//The Auction interface

  interface Auction
  {
    oneway void forAuction
    (
      in string itemName,
      in string auctioneerTopic,
      in string bidderTopic,
      in string observerTopic,
      in string bidderObserverTopic
    );
  };
```

The Auction interface defines the way in which information about the items available for auction is communicated between the auctioneer and the bidders or observers of an auction. Each message contains information relating to the auction of a single item, including:

- The name of the item.
- The topic on which bids can be made for the item.
- The topic on which the bidding is declared open and closed.
- The topic on which updates on the current bid are made.
- A topic to assist with a derived interface.

Bidders can be created and destroyed at any time. Therefore, this information needs to be preserved in much the same way that an auction catalogue preserves information on the items for auction. Messages sent to this interface should use the Store and Forward Protocol (`otsfp`).

The Auctioneer Interface

```
interface Auctioneer
  {
    oneway void bid
    (
      in float  bidValue,
      in string bidderName
    );
  };
```

The primary role of an auctioneer is to take bids on an item. No other communication from bidders to an auctioneer is required. Because there is a separate auctioneer for each item (to allow concurrent auctions to be managed by the same process easily), the only method required of the interface is the `bid()` method, which specifies the value of the bid, and the name of the bidder making it.

Because it is unlikely for bidders to be making bids while the auctioneer is doing other things, or is not available at all, the Auctioneer interface should use the Reliable Multicast Protocol.

The Observer and Bidder Interfaces

The auctioneer of an item makes two types of communication to the bidders:

- Informs bidders of the current value that has been bid for an item.
- Opens and closes an auction for an item.

Other components in the system can be interested in the bids being made on an item without needing to know when an auction starts and ends; these components are known as observers.

There is a separate interface for each type of communication; an `observer` interface and a `bidder:observer` interface. However, since a bidder for an item needs to know both when the auction starts and ends and the current bid on the item, it uses both interfaces. The `bidder` interface is, therefore, a derived interface of the `observer` interface. Messages sent on both interfaces should use the Reliable Multicast Protocol (`otrmp`).

```
interface Observer
  {
    oneway void currentBid
    (
      in float  bidValue,
      in string bidderName
    );
  };


  interface Bidder : Observer
  {
    oneway void auctionOpen();

    oneway void auctionClose
    (
      in float  bidValue,
      in string bidderName
    );
  };
```

Derived or polymorphic interfaces map well to an OrbixTalk Topic Name. For example, if messages to the `observer` interface use the following Topic Name:

```
otrmp//Auction/Mahogany Desk/Bidder/Observer/Messages
```

then messages to the `bidder` interface could be sent on the Topic Name:

```
otrmp//Auction/Mahogany Desk/Bidder/Messages
```

so that an application using the `bidder` interface can receive messages sent specifically to the `bidder` interface, or to its base interface, `observer`, by listening on the following wildcard Topic Name:

```
otrmp//Auction/Mahogany Desk/Bidder/**
```

The `bidderObserverTopic` string in the `Auction` interface is used to specify the wildcard Topic Name on which an application, using the `bidder` interface, should listen to receive messages sent on the `observer` interface.

OTAuction Topic Names

The following classes of Topic Names are used in the OTAuction system:

- The Auction Topic Name

  The Auction Topic Name is the topic on which information about the items available for auction is maintained. The Topic Name uses the Store and Forward Protocol (`otsfp`) so that the information is held persistently, and provided to any process that starts listening on the topic in the future; that is, the information is independent of time. There is one auction Topic Name per auction.

  It has the form:

  ```
  otsfp//<auction name>
  ```

  where `<auction name>` is the name of the auction.

- The Auctioneer Topic Name

  For each item that is auctioned, a Topic Name is required on which bids can be taken by the auctioneer. This is the auctioneer Topic Name. The auctioneer Topic Name uses the Reliable Multicast Protocol because bids do not need to be stored for future replay to the auctioneer.

  It has the form:

  ```
  otrmp//<auction name>/<item name>
  ```

  where `<item name>` is the name of the item being auctioned on this Topic Name.

  The `<auction name>` part is maintained because two separate auctions can exist for different items with the same item name.

- The Observer Topic Name

  The auctioneer for each item needs to inform each observer about the current bid value on the item. This is the observer Topic Name. The observer Topic Name uses the Reliable Multicast Protocol because the information is dependent on time, and has the form:

  ```
  otrmp//<auction name>/<item name>/bidding/bidder/
  observer
  ```

- The Bidder Topic Name

  The auctioneer also needs to inform bidders when the auction opens and closes. This is the bidder Topic which uses the Reliable Multicast Protocol (`otrmp`) because the information is dependent on time, and has the form:

  ```
  otrmp//<auction name>/<item name>/bidding/bidder
  ```

- The BidderObserver Topic Name

  Because bidders are also observers, and for ease of implementation reasons, a bidderobserver Topic Name is defined. It is a wildcard Topic Name that matches the bidder and observer Topic Name for a particular item. It has the form:

  ```
  otrmp//<auction name>/<item name>/bidding/*
  ```

Both the Reliable Multicast Protocol and Store and Forward Protocol guarantee ordering of messages, so the message stream of each protocol contains ordering information, which is related to the time at which messages are sent. The Auction topic uses the Store and Forward Protocol because the information is not dependent on time and guaranteed message delivery is required.

Similarly, those topics that use the Reliable Multicast Protocol in this example transport information that is dependent on time; for example, a bid must be made before an auction is closed. Since OrbixTalk does not guarantee that messages sent on separate Topic Names arrive in the same order, only those on an individual Topic Name, some components in the system require further checks to ensure that messages are handled correctly depending on the time at which they are received.

OTAuction Components

Auctioneer Program

The Auctioneer program sends a single message to describe each item that is available for auction, and creates an `Auctioneer_i` object to handle the actual auction of the item. The `Auctioneer_i` objects implement the `Auctioneer` interface, and talk on the `Observer` and `Bidder` interfaces through the relevant topics.

Each item that is auctioned has a specified reserve price. On receipt of a bid message, an `Auctioneer_i` object checks if the value of the bid is greater than the previous greatest value. If so, and the bidding has reached the reserve price, the `Auctioneer_i` object resets a timer that is used to close the auction. If no more bids greater than the largest bid are received before the timer fires, the auction is closed by sending an `auctionClose` message on the `Bidder` interface.

Bidder Program

Each auctioneer process manages the auctioning of items in a single auction that is referenced by the auction Topic Name associated with the auction name. Bidders that want to participate in a particular auction listen on the relevant auction Topic Name to determine the items that are available. For each `forAuction` message received on the auction Topic Name, a bidder process creates a `Bidder_i` object to handle the bidding for that item.

In this implementation, each `Bidder_i` object has a `bidLimit` and a `bidPause`. The `bidLimit` represents the greatest amount that the `Bidder_i` object is prepared to bid for an item. The `bidPause` represents the amount of time the `Bidder_i` object waits before making another bid, after hearing of a competing bidder leading the bidding for the item.

Since each `Bidder_i` object is interested in the `Bidder` interface, including those messages sent on the Observer Topic Name, it receives messages on the `bidderobserver` wildcard Topic Name to get messages sent on both the `observer` and `bidder` Topic Names for a particular item.

Running the Auctioneer and Bidder Programs

A typical run of OTAuction requires a number of bidder processes and one or more auctioneer processes. Both the OrbixTalk Directory Enquiries daemon (`otd/otdsm`) and the OrbixTalk MessageStore daemon (`otmsd`) should be started and be primary processes before starting the bidder or auctioneer processes.

Both programs can be run without any arguments to print out information about the expected arguments. Running the bidder without arguments produces the following output:

```
Usage: bidder <auction name> <bidder name> <bid limit> <bid pause>
```

Running auctioneer without arguments produces the following output:

```
Usage: auctioneer <auction name>
```

Running the Auctioneer Program

Each auctioneer process auctions items in a single auction. The name of this auction is specified in the `<auction name>` argument to auctioneer. The program prompts for a number of items to be auctioned, the name and reserve price for each of the items. You can wait for all bidders to be ready for the auction before pressing ENTER to continue. From this point, no further interaction is required.

Running the Bidder Program

Each bidder process makes bids on items in a single auction. The auction in which the bidder participates is specified in the first parameter. The remaining parameters to `bidder` are:

Parameter	Description
`<bidder name>`	Every bidder in an auction requires a unique name to distinguish its bids from that of other bidders. This parameter is also used as the persistent application name required by OrbixTalk for a listener using the Store and Forward Protocol (`otsfp`), so must be in the form of a valid application name. For example: `//bidder/1`
`<bid limit>`	This parameter specifies the limit to the bids that the bidder makes for every item in the auction. It should have a value between 1 and the maximum integer value.
`<bid pause>`	Before making a higher bid, each bidder waits a period of time after hearing of a competing bidder leading the bidding. This parameter specifies that time in milliseconds. It should be a value between 1 and the maximum integer value. It is interesting to note that a bidder with a greater value for this parameter tends to win fewer auctions where its competing bidders have the same bid limit.

Table 17.2: *Bidder Parameters*

Typical Output for the OTAuction Demonstration Program

Running the OTAuction demonstration program produces output similar to the following:

Auctioneer output

```
bash-2.00$ ./auctioneer MyAuction
*** Initialising OrbixTalk ***
How many items are for auction? 2
Item name    : Painting
Reserve price: 5.0
Item name    : Desk
Reserve price: 17.0
All items are ready for auction.  To begin the auctions,
press <enter>...
Opening auction for Painting on otrmp//MyAuction/Painting/
auctioneer
Opening auction for Desk on otrmp//MyAuction/Desk/auctioneer
Auction will begin now.
Received bid of 1 for Painting
This is the highest bid
Received bid of 1 for Painting
Received bid of 1 for Desk
This is the highest bid
Received bid of 1 for Desk
Received bid of 2 for Painting
This is the highest bid
Received bid of 2 for Desk
This is the highest bid
Received bid of 3 for Painting
This is the highest bid
Received bid of 3 for Desk
This is the highest bid
Received bid of 4 for Painting
This is the highest bid
Received bid of 4 for Desk
This is the highest bid
Received bid of 5 for Painting
This is the highest bid
Received bid of 5 for Desk
This is the highest bid
Received bid of 6 for Painting
```

```
This is the highest bid
We have reached reserve price
Received bid of 6 for Desk
This is the highest bid
Received bid of 7 for Painting
This is the highest bid
We have reached reserve price
Received bid of 7 for Desk
This is the highest bid
Received bid of 8 for Painting
This is the highest bid
We have reached reserve price
Received bid of 8 for Desk
This is the highest bid
Received bid of 9 for Painting
This is the highest bid
We have reached reserve price
Received bid of 9 for Desk
This is the highest bid
Received bid of 10 for Painting
This is the highest bid
We have reached reserve price
Received bid of 10 for Desk
This is the highest bid
Painting going 1 times...
Desk going 1 times...
Painting going 2 times...
Desk going 2 times...
Painting going 3 times...
Desk going 3 times...
Painting sold.
Painting was sold for 10 to //Bidder/2
Desk sold.
Desk was passed in
*** Terminating OrbixTalk ***
```

Bidder (1) output

```
bash-2.00$ ./bidder MyAuction //Bidder/1 10.0 500
*** Initialising OrbixTalk ***
Bidder is ready
New item available: Painting on topic
otrmp//MyAuction/Painting/auctioneer
```

```
New item available: Desk on topic otrmp//MyAuction/Desk/auctioneer
Bidding is open for Painting
Bidding is open for Desk
Bidding 1 for Painting
We hold the highest bid (1) on Painting
Bidding 1 for Desk
We hold the highest bid (1) on Desk
//Bidder/2 holds the highest bid (2) on Painting
//Bidder/2 holds the highest bid (2) on Desk
Bidding 3 for Painting
We hold the highest bid (3) on Painting
Bidding 3 for Desk
We hold the highest bid (3) on Desk
//Bidder/2 holds the highest bid (4) on Painting
//Bidder/2 holds the highest bid (4) on Desk
Bidding 5 for Painting
We hold the highest bid (5) on Painting
Bidding 5 for Desk
We hold the highest bid (5) on Desk
//Bidder/2 holds the highest bid (6) on Painting
//Bidder/2 holds the highest bid (6) on Desk
Bidding 7 for Painting
We hold the highest bid (7) on Painting
Bidding 7 for Desk
We hold the highest bid (7) on Desk
//Bidder/2 holds the highest bid (8) on Painting
//Bidder/2 holds the highest bid (8) on Desk
Bidding 9 for Painting
We hold the highest bid (9) on Painting
Bidding 9 for Desk
We hold the highest bid (9) on Desk
//Bidder/2 holds the highest bid (10) on Painting
That's too high for us.
//Bidder/2 holds the highest bid (10) on Desk
That's too high for us.
Bidding is open for Painting
//Bidder/2 holds the highest bid (10) on Painting
That's too high for us.
Bidding is open for Desk
//Bidder/2 holds the highest bid (10) on Desk
That's too high for us.
Bidding is open for Painting
//Bidder/2 holds the highest bid (10) on Painting
```

```
That's too high for us.
Bidding is open for Desk
//Bidder/2 holds the highest bid (10) on Desk
That's too high for us.
Bidding is open for Painting
//Bidder/2 holds the highest bid (10) on Painting
That's too high for us.
Bidding is open for Desk
//Bidder/2 holds the highest bid (10) on Desk
That's too high for us.
//Bidder/2 won the auction for Painting at 10
Nobody won the auction for Desk at 10
*** Terminating OrbixTalk ***
```

Bidder (2) output

```
bash-2.00$ ./bidder MyAuction //Bidder/2 10.0 500
*** Initialising OrbixTalk ***
Bidder is ready
New item available: Painting on topic
otrmp//MyAuction/Painting/auctioneer
New item available: Desk on topic otrmp//MyAuction/Desk/auctioneer
Bidding is open for Painting
Bidding is open for Desk
Bidding 1 for Painting
//Bidder/1 holds the highest bid (1) on Painting
Bidding 1 for Desk
//Bidder/1 holds the highest bid (1) on Desk
Bidding 2 for Painting
We hold the highest bid (2) on Painting
Bidding 2 for Desk
We hold the highest bid (2) on Desk
//Bidder/1 holds the highest bid (3) on Painting
//Bidder/1 holds the highest bid (3) on Desk
Bidding 4 for Painting
We hold the highest bid (4) on Painting
Bidding 4 for Desk
We hold the highest bid (4) on Desk
//Bidder/1 holds the highest bid (5) on Painting
//Bidder/1 holds the highest bid (5) on Desk
Bidding 6 for Painting
We hold the highest bid (6) on Painting
Bidding 6 for Desk
```

```
We hold the highest bid (6) on Desk
//Bidder/1 holds the highest bid (7) on Painting
//Bidder/1 holds the highest bid (7) on Desk
Bidding 8 for Painting
We hold the highest bid (8) on Painting
Bidding 8 for Desk
We hold the highest bid (8) on Desk
//Bidder/1 holds the highest bid (9) on Painting
//Bidder/1 holds the highest bid (9) on Desk
Bidding 10 for Painting
We hold the highest bid (10) on Painting
Bidding 10 for Desk
We hold the highest bid (10) on Desk
We hold the highest bid (10) on Painting
We hold the highest bid (10) on Desk
We hold the highest bid (10) on Painting
We hold the highest bid (10) on Desk
We hold the highest bid (10) on Painting
We hold the highest bid (10) on Desk
We won the auction for Painting at 10
Nobody won the auction for Desk at 10
*** Terminating OrbixTalk ***
```

The Source Code

The full source code for the OTAuction demo can be found in the OrbixTalk installation. The code includes many comments that assist your understanding of OrbixTalk programming in general and the specific situation for OTAuction.

Appendix E
OrbixTalk Class Reference

This appendix introduces the OrbixTalk Classes. These are used in the OrbixTalk API.

Appendix D, "Using the OrbixTalk API Directly" describes how to develop OrbixTalk applications using the OrbixTalk API as an alternative to the Event Service. This appendix provides a reference to classes used in that API.

In discussing the OrbixTalk API, *suppliers* are referred to as *talkers*, and *consumers* are called *listeners*.

Overview

The OrbixTalk Classes provide the following additional OrbixTalk functions that can be included in applications using the OrbixTalk API directly:

- Class `OrbixTalk` defines the interface to OrbixTalk. It includes functions to initialize OrbixTalk and to register and unregister talkers and listeners.
- Class `OrbixTalk::TimerEvent` is an abstract base class which defines the interface for timer events. When a timer event is fired, a talker or listener application receives a call-back and can periodically regain control from the OrbixTalk event loop.

  You can define a subclass of `OrbixTalk::TimerEvent` as a timed call-back to be called from the OrbixTalk timer service.

OrbixTalk Class

Description Class OrbixTalk defines the user interface to OrbixTalk.

Synopsis `// C++`

```
//
class OrbixTalk
{
public:
  enum REPLAY_TYPE
  {
    REPLAY_NONE      = 0,  // Do not replay messages
    REPLAY_ALL       = 1,  // Replay all messages - Default
    REPLAY_LAST      = 2   // Replay last message missed
  };
  class TimerEvent
  {
    // See entry for OrbixTalk::TimerEvent
  };
  static OrbixTalk_ptr initialise
  (
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
  virtual void terminate
  (
    const unsigned char bImmediate  = 0
  );
  virtual void registerListener
  (
    CORBA(Object_ptr)   pObject,
    REPLAY_TYPE         replayStore = REPLAY_ALL,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
  virtual CORBA(Object_ptr) registerTalker
  (
    const char*         pServerName,
    const char*         pTypeName,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
  virtual void registerTalker
  (
    CORBA(Object_ptr)   pObject,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
  virtual void unregister
  (
    CORBA(Object_ptr)   pObject,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
```

```
  );
  virtual unsigned char OrbixTalk::isRegistered
  (
    CORBA(Object_ptr)   pObject,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
  virtual void setPersistentAppName
  (
    const char*         pName,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
virtual void setMyReqTransformer
  (
    CORBA(IT_reqTransformer)*  pObject,
    CORBA(Environment)& rEnv = CORBA(default_environment)
  );
virtual CORBA(IT_reqTransformer)* getMyReqTransformer
  (
    CORBA(Environment)& rEnv
  );
  virtual unsigned char isIdle
  (
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
  virtual void addTimerEvent
  (
    TimerEvent*         pTimer,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
  virtual void removeTimerEvent
  (
    TimerEvent*         pTimer,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
};
```

OrbixTalk::addTimerEvent()

Synopsis

```
virtual void addTimerEvent
  (
    TimerEvent*         pTimer,
```

```
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Installs a timed call-back into the OrbixTalk timer service. When the timer expires, OrbixTalk::TimerEvent::fired() is called by the OrbixTalk timer service.

Parameters:

`pTimer`	A derived class of `OrbixTalk::TimerEvent` that implements a `fired()` method.

Notes: There is no guarantee that multiple timer events will not be fired simultaneously.

OrbixTalk specific.

See Also: `OrbixTalk::TimerEvent`
`OrbixTalk::TimerEvent::fired()`
`OrbixTalk::removeTimerEvent()`

OrbixTalk::isIdle()

Synopsis

```
virtual unsigned char isIdle
  (
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Determines when internal OrbixTalk message queues are empty, and OrbixTalk has finished internal processing.

Notes OrbixTalk specific.

See Also: `OrbixTalk::terminate()`

OrbixTalk::isRegistered()

Synopsis

```
virtual unsigned char OrbixTalk::isRegistered
  (
    CORBA(Object_ptr)   pObject,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Tests whether pObject is registered as an OrbixTalk talker or listener.

Parameters

`pObject`	The OrbixTalk talker/listener to be tested.

Notes OrbixTalk specific.

See also: `OrbixTalk::registerTalker()`
`OrbixTalk::registerListener()`
`OrbixTalk::unregister()`

OrbixTalk::initialise()

Synopsis

```
static OrbixTalk_ptr initialise
  (
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Initializes OrbixTalk. This function must be called by all OrbixTalk applications (including those that use the CORBA Event Service) before any interaction with Orbix or OrbixTalk.

Notes OrbixTalk specific.

See Also `OrbixTalk::terminate()`

OrbixTalk::registerListener()

Synopsis

```
virtual void registerListener
  (
    CORBA(Object_ptr)   pObject,
    REPLAY_TYPE         replayStore = REPLAY_ALL,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Registers an existing Orbix proxy object as an OrbixTalk listener.

Parameters

`pObject`	The Orbix object to be registered as a listener.

relayStore	For an otsfp protocol, determines how messages are replayed from the MessageStore: REPLAY_NONE = 0 No replay. REPLAY_ALL = 1 Replay all messages not yet heard. REPLAY_LAST = 2 If messages are missed, replay the most recent.

Notes OrbixTalk specific.

OrbixTalk::registerTalker()

Synopsis

```
virtual CORBA(Object_ptr) registerTalker
  (
    const char*         pServerName,
    const char*         pTypeName,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Creates a proxy object for the IDL interface specified in pTypeName and registers it as an OrbixTalk talker on the topic specified in the pServerName argument.

Subsequent invocations on the proxy use the OrbixTalk transport layer. Only oneway operations can be invoked on the proxy.

Parameters

pServerName	This parameter is in one of the following forms: myMarker Object is created using the marker "myMarker", and registered on the otrmp//myMarker topic. myMarker:otrmp//myTopic Object is created using the marker "myMarker" and registered on the "otrmp//myTopic" topic. otrmp//myTopic Object is created using a marker supplied by Orbix, and registered on the "otrmp//myTopic" topic.
pTypeName	This parameter specifies the interface for which the proxy is created.

Note: In the above examples, otrmp can be any valid OrbixTalk protocol type.

Notes OrbixTalk specific.

See Also OrbixTalk::unregister()
OrbixTalk::isRegistered()

OrbixTalk::registerTalker()

Synopsis

```
virtual void registerTalker
  (
    CORBA(Object_ptr)   pObject,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Registers an existing proxy object as an OrbixTalk talker. A typical use is to register a proxy created following a look-up using the Naming Service.

Subsequent invocations on the proxy will use the OrbixTalk transport layer. Only oneway operations can be invoked on the proxy.

Parameters

`pObject`	A pointer to an Orbix proxy object.

Notes OrbixTalk specific.

See Also `OrbixTalk::unregister()`
`OrbixTalk::isRegistered()`

OrbixTalk::removeTimerEvent()

Synopsis

```
virtual void removeTimerEvent
  (
    TimerEvent*         pTimer,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Removes an un-expired timer from the OrbixTalk timer service.

Parameters

`pTimer`	A pointer to an instance of an object derived from `OrbixTalk::TimerEvent`.

Notes OrbixTalk specific.

See also `OrbixTalk::TimerEvent`
`OrbixTalk::addTimerEvent()`

OrbixTalk::setPersistentAppName()

Synopsis

```
virtual void setPersistentAppName
  (
    const char*         pName,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description All OrbixTalk listeners using the OrbixTalk Store and Forward Protocol must have a unique application name that is set using this function. Talkers using the otsfp can also set a persistent application name using this function, allowing them to maintain state across process invocations.

Parameters

`pName`	The application name for the OrbixTalk talker or listener. This name should be in a format similar to that shown below: `//Part1/Part2/Part3` where any number of parts can be used in the name.

Notes This function must be called after `OrbixTalk::initialise()`, and before any talker or listeners are registered. The name should not change between invocations of the application.

OrbixTalk specific.

OrbixTalk::setMyReqTransformer

Synopsis

```
virtual void setMyReqTransformer
  (
    CORBA(IT_reqTransformer)*  pObject,
    CORBA(Environment)& rEnv           =
CORBA(default_environment)
  );
```

Description Registers an `IT_reqTransformer` object as the default transformation for requests leaving or entering the address space via the OrbixTalk transport.

Parameters

`pObject`	A pointer to the transformer object.

Notes The transformer object has the same type as the transformers used in Orbix. Tranformers set using this operation are specific to requests sent or received using the OrbixTalk transport. The transformer applies to all requests, regardless of whether the calling and target objects are co-located or not.

In OrbixTalk, all communication is connectionless, so the `setRemoteHost()` operation in the `CORBA::IT_reqTransformer` class is redundant. When using transformers, the data passed into the transformer does not need to be deleted if new data is put in its place.

For more information about using transformers, refer to the *Orbix Programmer's Reference*.

See also `getMyReqTransformer`

OrbixTalk::getMyReqTransformer

Synopsis

```
virtual CORBA(IT_reqTransformer)* getMyReqTransformer
  (
    CORBA(Environment)& rEnv
  );
```

Description Returns a pointer to the `IT_reqTransformer` object that was registered using `setMyReqTransformer`. If no transformer has been registered, a null pointer is returned.

Notes For more information about using transformers, refer to the *Orbix Programmer's Reference*.

See also `setMyReqTransformer`

OrbixTalk::terminate()

Synopsis

```
virtual void terminate
(
   const unsigned char bImmediate  = 0
);
```

Description This function should be called before exiting an OrbixTalk application. It releases the resources associated with OrbixTalk and Orbix.

This function can only be called from the main application thread. Calling this function from within a timer thread can cause unpredictable behavior. The recommended approach is to set a flag in the timer thread and catch this from the main application thread.

Parameters

`bImmediate`	This parameter defaults to 0, for which the function will not return until OrbixTalk has finished internal processing, ensuring that pending messages are sent correctly, and that incoming messages are dispatched correctly. When set to 1, the function returns immediately.

Notes OrbixTalk specific.

OrbixTalk::unregister()

Synopsis

```
virtual void unregister
  (
    CORBA(Object_ptr)   pObject,
    CORBA(Environment)& rEnv        = CORBA(default_environment)
  );
```

Description Unregisters a talker or listener object, preventing a listener from having incoming messages dispatched to it, and a talker from making further invocations.

Parameters

`pObject`	A pointer to an OrbixTalk talker or listener.

Notes OrbixTalk specific.

OrbixTalk::TimerEvent Class

Synopsis Class `OrbixTalk::TimerEvent` is an abstract base class that defines the interface for timed callbacks. The class is used by defining a subclass, implementing the `fired()` method within it, and inserting an instance of the subclass in the OrbixTalk timer service using the `OrbixTalk::addTimerEvent()` function.

Orbix

```
class OrbixTalk::TimerEvent
{
  public:
    virtual void fired() = 0;
  protected:
    virtual ~TimerEvent();
    TimerEvent(const unsigned int timeout);
    unsigned int setTimeout(const unsigned int timeout);
    unsigned int getTimeout();
  private:
    unsigned int m_milliSecDelay;
    void*        m_dummy1;
}
```

Notes OrbixTalk specific.

OrbixTalk::TimerEvent::TimerEvent()

Synopsis

```
TimerEvent
  (
    const unsigned int timeout
  );
```

Description Constructs an instance of an `OrbixTalk::TimerEvent`.

Parameters

`timeout`	Specifies the number of milliseconds after which the OrbixTalk timer service calls the `fired()` method of the `TimerEvent`.

Notes `OrbixTalk::TimerEvent` objects fire once. If an application requires a repeating timer, the timer should add itself to the OrbixTalk timer service in its `fired()` method.

OrbixTalk specific.

OrbixTalk::TimerEvent::fired()

Synopsis

```
virtual void fired() = 0;
```

Description This function is called when the `OrbixTalk::TimerEvent` is fired by the OrbixTalk timer service.

Notes `OrbixTalk::TimerEvent` objects fire once. If an application requires a repeating timer, the timer should add itself to the OrbixTalk timer service in its `fired()` method.

OrbixTalk specific.

OrbixTalk::TimerEvent::getTimeout()

Synopsis

```
unsigned int getTimeout();
```

Description Determines the timeout interval after which the timer would fire if inserted into the OrbixTalk timer service. This does not return the remaining interval, but the original interval specified in the constructor.

Notes OrbixTalk specific.

OrbixTalk::TimerEvent::setTimeout()

Synopsis

```
unsigned int setTimeout
  (
    const unsigned int timeout
  );
```

Description Sets the interval after which the timer will fire.

Parameters

`timeout`	Specifies the timeout interval in milliseconds.

Notes If the timer is already in the OrbixTalk timer service, after calling this function, the timer fire after timeout milliseconds from the time the function is called. If the timer is not in the OrbixTalk timer service, when added it fires after timeout milliseconds from the time it is added.

OrbixTalk specific.

Index

L

M

N

O

P

R

S

T

U

V