

Implementation of the CORBA Event Service in Java

A thesis submitted to the
University of Dublin, Trinity College,
in fulfilment of the requirements for the degree of
Masters of Science (Computer Science).

Paul Stephens,
Department of Computer Science,
Trinity College, Dublin.

September 1998.

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this or any other University, and that, unless otherwise stated, it is entirely my own work.

Paul Stephens,
September 1998.

Permission to Lend and/or Copy

I, the undersigned, agree that the Trinity College Library may lend and/or copy this thesis upon request.

Paul Stephens,
September 1998.

Summary

The increase need and interest in distributed technology has led to several different types of object orientated middleware. CORBA is being defined by the Object Management Group (OMG) and is the most commonly used middleware today for building distributed applications. One of the fundamental requirements for a distributed system is to communicate event information between objects. The CORBA standard defines a specification of an Event Service that provides a flexible model for asynchronous communication among objects. This thesis describes the CORBA standard briefly and gives a detail account of the Event Service. The CORBA Event Service specification lacks important features required by application domains such as telecommunications and real-time systems. These limitations are described along with the OMG proposals for a Notification Service and Messaging Service that will overcome the various drawbacks. Also described is a comparison of various event models both CORBA and non-CORBA based, that have been developed by commercial companies and academic institutions.

This thesis describes the requirements, design and implementation of TCDEvents, a Java implementation of the CORBA Event Service. Finally, the evaluation of TCDEvents is compared with other CORBA and non-CORBA event models.

Acknowledgements

First and foremost, thanks to my supervisor, Dr. Paddy Nixon. I would also like to thank Samantha Murphy for supporting my work, and having to live with me during the past year.

Thanks also to Rene Meier for helping to proof read this script.

Finally, thanks to my family, friends and colleagues for their support.

Contents

Chapter 1 Introduction	7
Distributed Object Technology	7
An Overview of CORBA	9
The Basic CORBA Communication Paradigm	13
Services and Application Areas.....	15
Expected Goals of the Project	17
Dissertation Overview.....	17
Chapter 2 The CORBA Event Service	18
Object Communication using the CORBA Event Service.....	18
Methods of Event Communication	21
The Push Model.....	21
The Pull Model.....	21
Mixing the Push and Pull Models in a Single system.....	22
Point to Point Events.....	25
Types of Event Communication	26
Untyped Event Communication	26
Typed Event Communication.....	26
Interfaces for the CORBA Event Service	27
The Push Event Interfaces.....	27
The Pull Event Interfaces	28
Typed Interfaces for the Push and Pull Models	29
The Administration Interfaces	29
The Event Service and Existing CORBA Services	31
The CORBA Time Service	31
The CORBA Security Service.....	33
Evaluation of the OMG COS Event Service	35
The Notification Service	38
CORBA 3.0 – The Next Generation	41
Summary	43
Chapter 3 Academic and Commercial Event Models	44
CORBA Event Models.....	44
Non-CORBA Event Models.....	51
Summary	54
Chapter 4 Requirements for TCDEvents	55
System Model.....	55
Functional Requirements	57
Additional Requirements	61
Use Case Models	62
Summary	64
Chapter 5 Design and Implementation of TCDEvents	65
Detail Design	66
Event Server	66
Registering the Event Service in the Implementation Repository.....	68
Publishing the Channel Identifiers with the Naming Service	69
Event Channel	72
Channel Manager.....	75
The Push Model for Untyped Events.....	76

The Pull Model for Untyped Events.....	79
Locating the Event Channel and Channel Manager	82
The Consumer/Supplier Wizard	84
Sequence Diagrams.....	87
Class Diagrams	91
Package Diagram	93
Summary	93
Chapter 6 Evaluation of TCDEvents.....	94
The Push Model Test	96
Average Times for Push Model Tests.....	101
Push Model Test Conclusions	101
The Pull Model Test	103
Average Times for Pull Model Tests.....	107
Pull Model Test Conclusions	107
Performance of the Push Model	108
Summary	110
Chapter 7 Conclusion.....	111
Chapter 8 Glossary	116
Chapter 9 References	117
Appendix A TCDEvents IDL Definitions.....	121
CosEventComm.idl	122
CosEventChannelAdmin.idl	123
EventServerAdmin.idl.....	124
Appendix B TCDEvents Source Code	125
TCDEvents Source code	126
GUI Tool Source Code	166

1

Introduction

Distributed Object Technology

To meet today's challenges of a rapid changing Information Technology (IT) business environment, organizations have to continually maintain competitive advantage, respond quickly to customer demands, and provide a product of a high quality at low cost. These demands on IT organizations are no different than they were, say twenty years ago. The only real difference now is that more and more organizations rely on technology to gain advantages over their competitors. With this, there is a new challenge for the IT organizations of the 90's; trying to keep up to date with the ever fast increases in technology.

Today, most IT organizations are under a considerable amount of pressure to deliver higher value products at lower costs, and within much shorter time to market schedules, adding to the ever increasing burden on organizations. Most are struggling to manage complex, heterogeneous environments that have different hardware, software, applications, networks and database systems. Many new systems that are developed today using the latest internet or client/server technology, must integrate with legacy systems that were originally developed ten or twenty years ago. Driven by increasing demands on an organization to bring products to market quicker, has resulted in many organizations adopting a tactical short term approach to satisfy increasing customer demands. This has resulted in the lack of organizations setting a long term technical direction, and just settling for the quick fix.

The past two decades have seen many changes in the way enterprise information systems get designed, developed and maintained. It all started with the monolithic mainframe systems. Each of these systems contained all of its own presentation,

business and data access logic. These systems could not share data with other systems, which meant that each system had to maintain its own private copy of the data. Since different systems needed access to the same data, organizations had to store redundant copies on multiple machines. These monolithic applications were inefficient and costly, they were soon replaced by relational database technology and the client/server model. Client/server computing promised to simplify large and complex systems by separating centralized systems into components that could be developed and maintained with ease. The client component would typically implement the presentation logic whilst the server would contain the business logic. Either the client or the server, depending on the strategy of the organization, would handle the data access logic. In the end, many of these solutions were just two separate monolithic systems. One of the major problems with these types of systems was that developers found it very difficult to reuse code. Development teams would create modules of similar functionality over and over again, to be used in different parts of an enterprise system. The problem occurred, that if a change was required in one module, that change would have to be propagated to all other modules containing the changed functionality, throughout the enterprise. The problem of managing the changes in an effective way arose, as functionality inconsistencies crept into the enterprise information system.

Distributed object technology, coupled with a powerful communication infrastructure fundamentally changes the problems inherent with client/server computing. The idea is to change monolithic client/server applications into self-managing objects, which can interoperate across different networks and operating systems. A distributed object computing model enables IT organizations to build an infrastructure that is more adaptive to ongoing change and responsive to market opportunities, making it easier for an organization to establish and maintain a competitive advantage. However, with this new model came new requirements, distributed business applications must be able to operate in a heterogeneous environment being able to operate on different hardware and software platforms. They must also integrate old and new technology and make use of existing infrastructures.

An Overview of CORBA

The Object Management Group (OMG) is a consortium of over 700+ companies that represent the entire spectrum of the computer industry. Since 1989 the main purpose of the OMG has been to define the Common Object Request Broker Architecture (CORBA). CORBA is a distributed object computing middleware standard that has been widely accepted by the computer industry. The notable exception is Microsoft, which has its own competing object broker called Distributed Component Object Model (DCOM). CORBA has been designed to support the development of flexible and reusable distributed services and applications. Its object orientated approach enables different types of implementations to interoperate. The interoperability is achieved with well defined interface specifications at application level. The language that is used by CORBA to provide portable interfaces is called Interface Definition Language (IDL). IDL provides no implementation details and defines the component boundaries.

In the autumn of 1990, the OMG published the first version of the Object Management Architecture Guide (OMA Guide) [OMG:93]. The architecture consists of four main elements as shown in Figure 1-1.

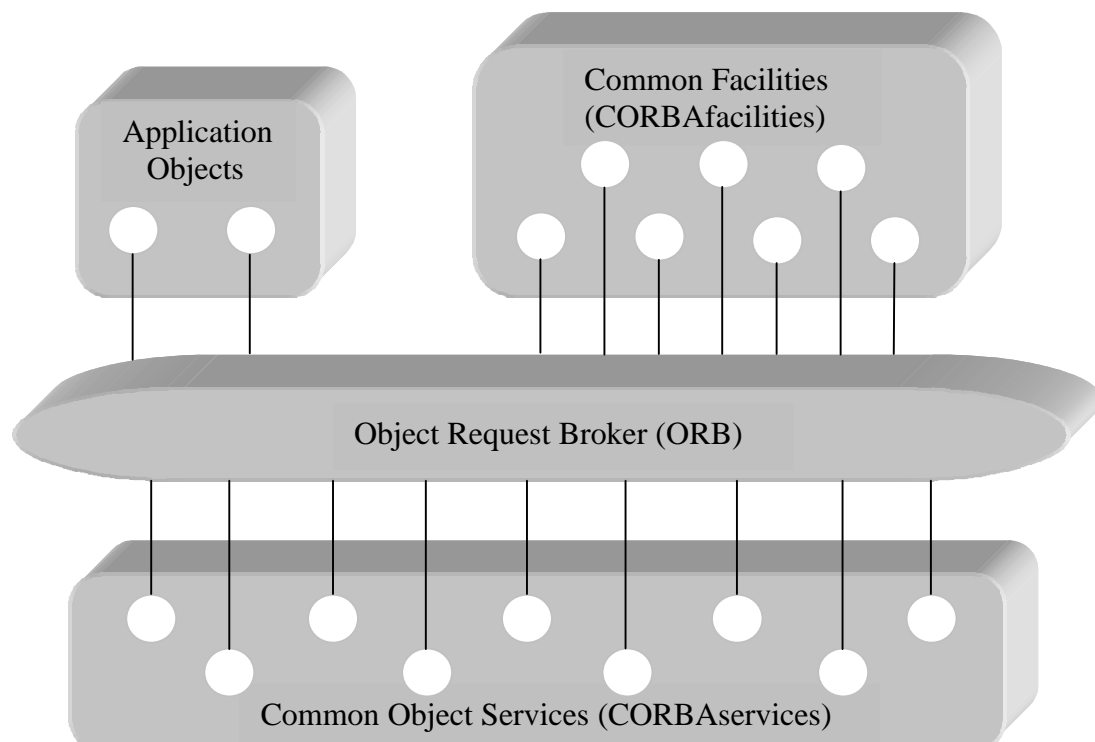


Figure 1-1 The OMG Object Management Architecture

1. Object Request Broker (ORB)

The ORB is the heart of the CORBA architecture. It is an object bus that allows communication between objects located locally or remotely. The CORBA ORB provides a wide variety of distributed middleware services. It allows objects to discover each other at runtime and invoke each other services. The ORB is much more sophisticated than other forms of Client/Server middleware such as Remote Procedure Calls (RPC) [BN:84], Message Oriented Middleware (MOM), database stored procedures and peer-to-peer services. CORBA provides the best middleware solution to date because of the benefits of the ORB. These benefits are; static and dynamic method invocations, high-level language bindings, self-describing systems, local/remote transparency, built in security and transactions, polymorphic messaging and coexistence with legacy systems. All these benefits and more are described in detail in [OMG:95a].

1. CORBA Services

CORBA Services provide developers with the ability to build distributed systems that are flexible and meet the needs of customers. These object services complement the functionality of the ORB, allowing developers to concentrate on the more complicated issues when building distributed systems, such as implementing their own application and business logic. The CORBA Event Service is specified along with other services in this area, which are fundamental for building distributed systems.

2. CORBA Facilities

These are collections of IDL defined frameworks that provide services for direct use with application objects. They provide richness and application-level focus to the ensemble of OMG technologies, extending the fundamental CORBA technologies up to application developer and independent software vendor level. There are two types of common facilities; Vertical Frameworks and Horizontal Frameworks. The Vertical Frameworks are for market specialty areas such as, health, retail, finance etc. The following is a list of CORBA based Vertical Frameworks that have been developed or are in the process of being developed; CORBA for Document Imaging, CORBA for Information Superhighways, CORBA for Computer Integrated Manufacture, CORBA for Distributed Simulations, CORBA for Oil and Gas Exploration, CORBA for Accounting etc. The Horizontal Frameworks are application domain independent, there are four different types; User Interface, Information Management, System

Management and Task Management. The OMG documents [OMG:95c] and [OMG:95d] explain in detail the CORBA facilities and their status.

3. Application Objects

These are the business objects and applications that are specific to the end user. A distributed system is typically built from a number of cooperating business objects that work together. Business objects are self-contained *deliverables* that have a user interface, a state, and know how to cooperate with other separately developed business objects to perform the desired task. They are variations of the Model-View-Controller (MVC) paradigm [JL:97].

The OMG and its members are aiming for a future in which software objects with defined interfaces interoperate across corporate Intranets and the Internet. The benefits that CORBA brings to application developers and IT organizations is significant :

- **Integration with Existing Systems**

One of the big problems in building distributed systems today is integration with legacy applications. Defining an IDL interface entry point to the legacy application gives it a CORBA compliant interface, allowing it to interoperate with other CORBA objects in the system.

- **Portability**

Any object that is CORBA compliant is portable. This means that objects developed on one platform can be deployed on any other platform that CORBA supports.

- **Plug and Play**

Every single CORBA compliant object has a defined IDL interface. Access to the object has to go through the interface. As with the object oriented paradigm, changes to the implementation of the object will not affect other objects, as long as the object's interface remains the same. This allows developers to modify objects, without breaking any other part of the distributed system.

- **Choice**

CORBA is an open solution based on published specifications that get implemented by CORBA vendors. CORBA is implemented and supported on a wide variety of hardware and operating system platforms, allowing IT

organizations to choose the platform that is best suited for their distributed system development.

- **Interoperability**

The OMG has specified the Internet Inter-ORB Protocol (IIOP) that allows communication between objects connected to different vendor ORB's. With the use of software bridges CORBA compliant objects and objects developed by Microsoft's ActiveX/DCOM can communicate. This enables IT organizations to select objects based on the functionality they provide regardless of the vendor.

The Basic CORBA Communication Paradigm

The basic CORBA communication paradigm between clients and servers is a one-to-one synchronous model, which is fundamental to the CORBA architecture [OMG:95a]. Figure 1-2 illustrates the basic CORBA model for communication between distributed applications.

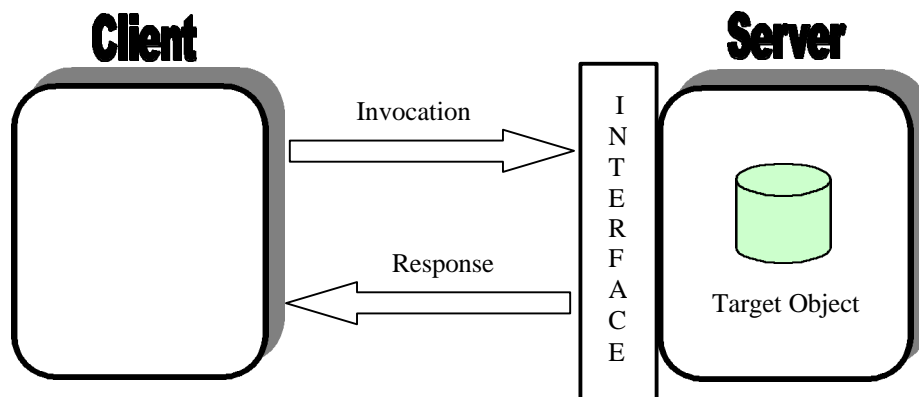


Figure 1-2 Basic CORBA Client-Server Communication Model

With this model, data is communicated by a client invoking operations on a server object by an invocation request. The invocation will normally contain data arguments that are marshaled and sent over the network to the server object. Once a client sends an invocation to a server, it is blocked and unable to perform any other function. The server side unmarshals the request to obtain the data, perform the necessary operations and returns a response back to the client. When the client has received the response it may continue its execution. For each invocation, there must be a single client and a single server available for the invocation call to succeed. If a request fails for any reason, for example the server is unavailable, clients will receive an exception and must take appropriate action. This is the same basic paradigm of all Remote Procedure Calls (RPC) based systems, such as Distributed Computing Environment (DCE).

In many cases, this basic communication paradigm is sufficient and greatly eases development since it extends the distributed computing in a straight forward manner from normal non-distributed function calls. However, in some scenarios a more decoupled communication model between clients and servers is required. Examples of this may include :

1. Clients do not know the identity of corresponding servers and therefore cannot bind to them directly. The server required to handle the request is chosen independently of the client.
2. Clients may wish to send requests to servers when they are not available. In this case, there needs to be a mechanism that will buffer requests until a server becomes active again.
3. Clients may have to be responsive to other events and cannot afford to block while waiting for a response from a server.

CORBA does offer standardized techniques beyond the basic communication paradigm that can help overcome some of the problems mentioned above. Such techniques are oneway method calls, DII deferred invocation, distributed callbacks [DSSV8:96] and the Event Service. Furthermore ORB vendors may offer non-CORBA standard services that also address these problems [RIR:96].

One of the ways that a developer can overcome some of the burden of the basic communication paradigm model is to utilize an implementation of the *OMG Events Service*. An *event* is an occurrence within an object, specified to be of interest to one or more objects. A *notification* is a message an object sends to interested parties informing that a specific event has occurred. The Event Service is one component in the OMG Common Object Services Specification (COSS) Volume 1 [OMG:95b]. Its purpose is to provide delivery of event data from suppliers to consumers without requiring these participants to know about each other explicitly. Implementations of the Event Service act as “mediators” that support decoupled communication between objects. Events are more loosely coupled than RPC but less loosely coupled than Message-Oriented Middleware (MOM) [OHE:96].

Services and Application Areas

As CORBA is maturing and becoming the industry recognized standard for building distributed systems, more new types of service are emerging along with the applications which access them:

- **Multimedia Services**

Broadband ISDN is expected to be the universal network of the future [FH:96]. It is able to supply bandwidth on demand up to very high speeds and to handle voice and new data types, such as; Speech and Music, Image, Business Video and Entertainment Video. Distributed systems will be expected to handle the Multimedia Services and their applications such as Hypermedia. The use of Event Management will play a critical role in this new service. For example, video conferencing that is supplied to the user's PC terminal. A user mouse clicks on the video presentation area may cause the video to be paused whilst the audio output still continues [JB:95].

- **Cooperative Systems**

Cooperative applications support collaboration between a number of users in a system. Events generated in one cooperative application have to be reflected in all other cooperative applications in the system.

- **Distributed Artificial Intelligence**

The merger of Artificial Intelligence (AI) with the Internet has resulted in Distributed AI. The first generation, the "intelligence" was localized but now the "intelligence" is distributed around the system. The communication between the intelligent entities is the key for a successful distributed AI system. Such AI systems could be used in Electronic Commerce and Network Management using mobile agents [FS:97].

- **Telecommunications Management**

Integration of the telecommunications world with the distributed computing world has already started and proposals for CORBA to be included in Telecommunication Management Network (TMN) [VS] and Telecommunication Information Network Architecture (Tina) [CM:95] have been made. Since CORBA objects are event driven between managers and agents, it is ideal for

monitoring telecommunication systems for fault and overload conditions. As well as benefiting from the other aspects of CORBA.

- **Real-time Systems**

There is a widespread belief among software developers of real-time systems that object orientated techniques are not suitable. Since the polymorphic behavior properties of object orientated programming are the direct opposite to real time systems that require deterministic behavior and low latency. However, many real time applications, such as avionics, process control, distributed interactive simulation etc, can benefit from the advantages that CORBA offers. An avionics mission control application described in [HSL:97] is based on CORBA and compares it with its non-CORBA equivalent.

- **Mobile Programming Environments**

Mobile programs can move from one host to another during execution. Adaptive mobility is extremely useful for collaborative computer applications and for querying distributed databases, because it allows programs to move in response to changes in available resources.

Mobile programming environments are becoming more useful in building distributed applications, particular for multi-user internet applications, such as Web Chat applications. In this case, Web Chat servers could move to different hosts based on changes on network latencies and client locations.

Within applications of these types, event driven operations are often the most natural paradigm. These systems often need to be informed of an event occurrence to initiate further action if required. OMG specifies a basic Event Service, but for many of these applications it is not suitable as it stands. The OMG are in the process of rectifying this with the soon to be introduced Notification Service and Messaging Service.

Expected Goals of the Project

This project will be a full implementation of the OMG CORBA Event Service that supports both, the push and the pull style communication between suppliers and consumers of events. It will be written purely in the Java programming language, taking advantage of the strengths of Java. It will also implement features that minimize resource utilization, including thread pool management and the ability to limit the number of events in transit.

The production of the Java Event Service will involve all stages of the software development lifecycle; specification, design, implementation and testing.

The project will also involve an investigation into the current problems of the OMG CORBA Event Service and the proposed solutions to overcome these problems. A comparison of various academic and commercial Event Service solutions (CORBA and non-CORBA) with the OMG solution and the different approaches in solving the problems of events in large scale distributed systems will also be discussed.

Dissertation Overview

This dissertation is divided up into 7 chapters.

Chapter 2 describes an overview of the OMG CORBA Event Service, the problems associated with it and solutions that the OMG are providing.

Chapter 3 discusses various academic and commercial event models both, CORBA and non-CORBA based.

Chapter 4 describes the requirements for TCDEvents, a Java implementation of the CORBA Event Service.

Chapter 5 describes in detail the full design and implementation of TCDEvents, it makes extensive use of the Unified Modeling Language (UML).

Chapter 6 describes the evaluation of TCDEvents compared to other CORBA and non-CORBA Event Services.

Chapter 7 summarizes the main findings, and describes the important features of the implementation.

2

The CORBA Event Service

The CORBA Event Service defines a model of communication that allows an application to send an event that will be received by any number of objects. This chapter describes in detail the CORBA Event Service, and how it fits in with the existing CORBA Services. It also discusses the drawbacks it has for many distributed applications and the proposals from the OMG to overcome these drawbacks.

Object Communication using the CORBA Event Service

The basic request-reply communication model is fundamental to the CORBA architecture. However, many distributed applications require a more complex, indirect communication mechanism. The CORBA Event Service defines a communication model that allows an application to send messages to objects, without even knowing that the target object that will receive the message exists.

The OMG has specified a CORBA Event Service [OMG:95b], which introduces the concept of events to the CORBA communication paradigm. The Event Service allows objects to dynamically register or un-register their interest to specific events. Objects can have one of two roles; *Suppliers* or *Consumers*, which are defined by the Event

Service. A supplier is the object that generates an event, and a consumer is the object that receives the event. The Event service decouples the communication between objects. A supplier has no knowledge of the number of consumers the event is being sent to, and consumers have no knowledge of the supplier that generated the event.

In order to support this type of operation, the CORBA Event Service has introduced a new CORBA architectural element called *Event Channel*. The Event Channel is an intervening object that can act as both, supplier and consumer of events. It allows multiple suppliers to communicate asynchronously with multiple consumers without knowing of their existence. Suppliers and consumers never connect directly to each other, but instead connect to the Event Channel. To the supplier the Event Channel looks like a consumer and to the consumer the Event Channel looks like a supplier, as shown in Figure 2-1. The Event Channel object sits on the ORB and is the key that allows decoupled communication between suppliers and consumers.

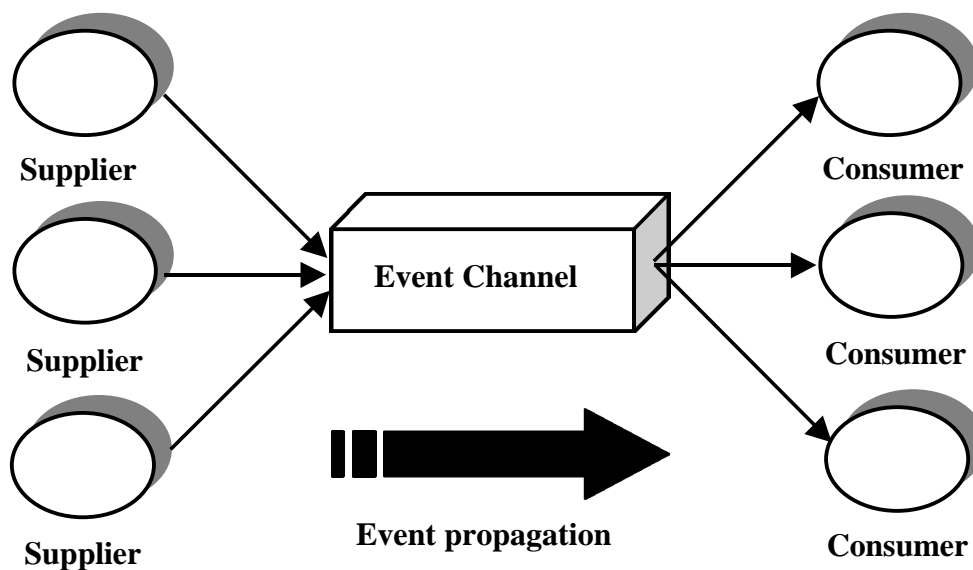


Figure 2-1 The Event Channel

There is no correlation between the number of suppliers and consumers. Suppliers and consumers can be added to the Event Channel with ease without having any impact on the system. Suppliers and consumers are also not restricted to being connected to a single Event Channel; they can be connected to as many Channels as feasibly possible.

A typical example that may need an event-based system is a spreadsheet application. Suppose a cell in the spreadsheet is linked to a number of different documents. If the cell gets updated, the updates will need to get propagated to the documents, without the spreadsheet software knowing which documents are linked to the cell. Using the CORBA Event Service this can be solved very easily, the cell will just issue an event every time it gets updated. The documents will then be able to receive the event and update their properties as required. Another advantage of the loosely coupled system, is documents can be added and removed without effecting the spreadsheet software.

Methods of Event Communication

The CORBA Event Service specifies two basic models for allowing suppliers and consumers to communicate event data; the *push* model and the *pull* model.

The Push Model

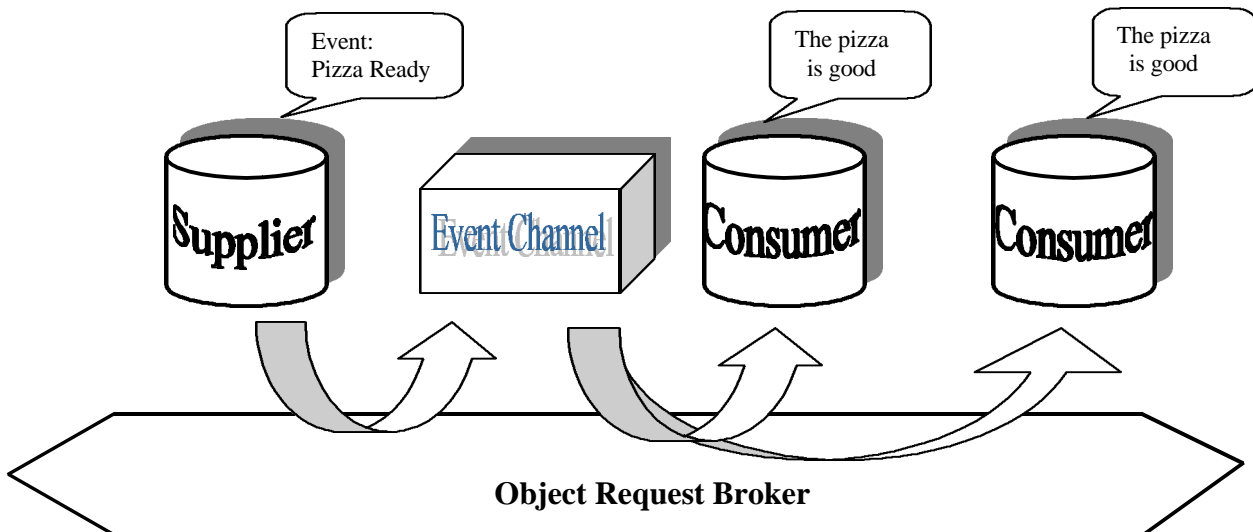


Figure 2-2 Push Style

An active supplier issues an event making a push method invocation. The event gets passed to the Event Channel object who plays the role of *Notifier*, and is responsible for passing it onto all the passive consumers that are registered (connected) to the channel. In this model suppliers can be thought of as clients and consumer as servers.

The Pull Model

The pull model see Figure 2-3, is the reverse to the push. An active consumer makes a pull request to the Event Channel to get event data, the channel playing the role of *Procurer* will then pull the event from the passive supplier and issues it to the consumer.

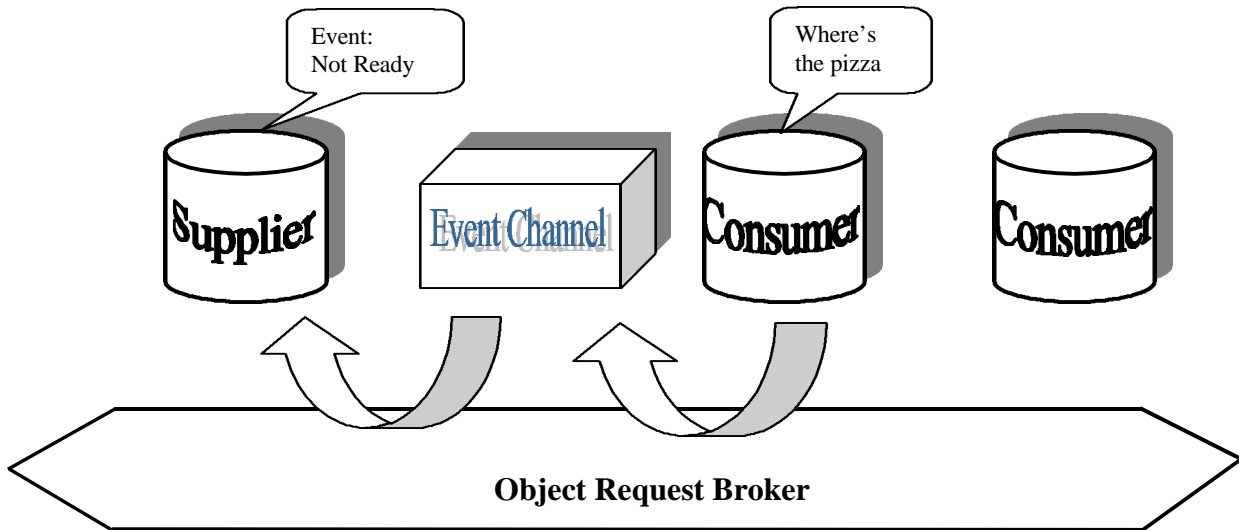


Figure 2-3 Pull Style

In this model consumers can be thought of as clients and suppliers as servers.

Mixing the Push and Pull Models in a Single system

Since the suppliers are decoupled from the consumers by the Event Channel, it is possible to have a mixture of both models in an event system. Consider, a hybrid configuration as in Figure 2-4, suppliers use the push model and consumers use the pull model, this is commonly known as the Push/Pull model.

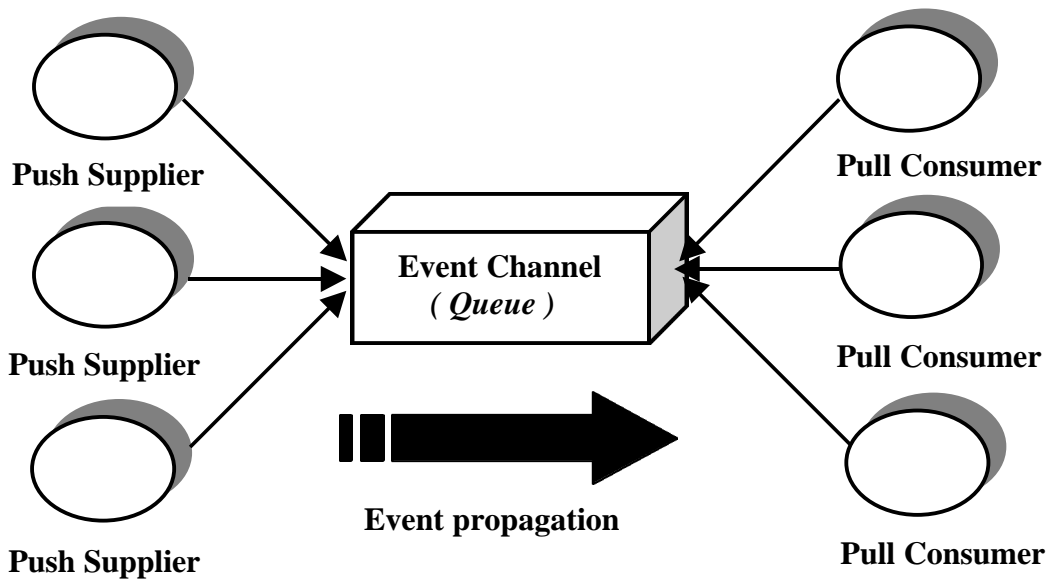


Figure 2-4 Push/Pull Model

In this case both, suppliers and consumers could initiate the event transfer. Suppliers invoke an operation on an object in the Event Channel to transfer event data. Consumers would invoke another operation on the object in the Event Channel to obtain event data. The Event Channel takes no initiative in forwarding the events and can be thought of as playing the role of a *Queue*. It will store events supplied by push suppliers until some pull consumers make requests for an event.

The Pull/Push model shown in Figure 2-5 is another hybrid that allows the Event Channel to pull events from passive suppliers and push them to passive consumers. The Event Channel plays the role of *Intelligent agent*. Having an active Event Channel means that it can pull data from passive suppliers and push event data to passive consumers.

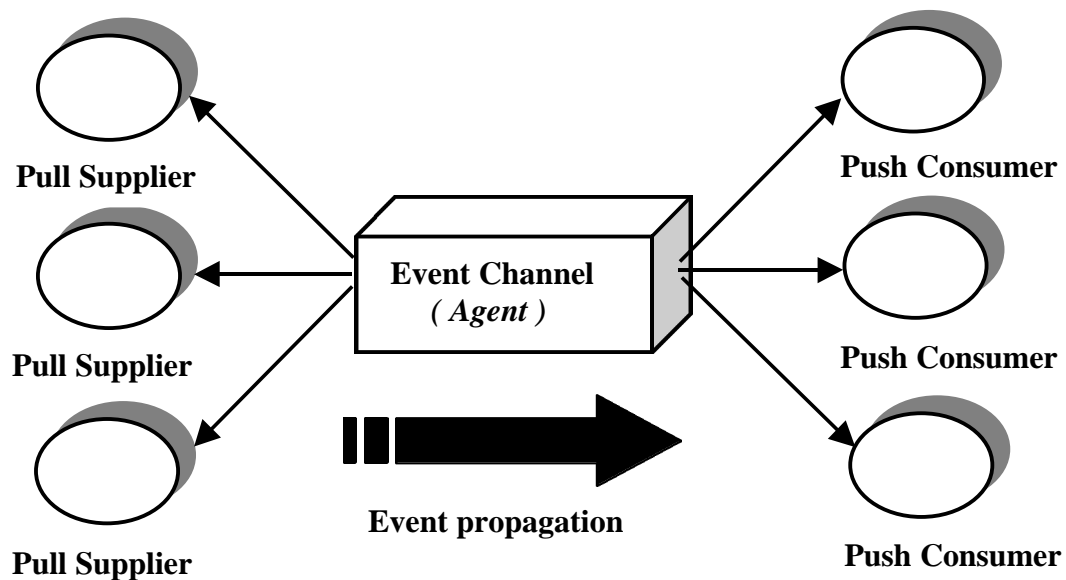


Figure 2-5 Pull/Push Model

Suppliers and Consumers that are registered with the Event Channel are not restricted to being of the same type. For example, suppliers connected to the Event Channel can be a mixture of both push and pull models. The same applies to consumers, as shown in Figure 2-6.

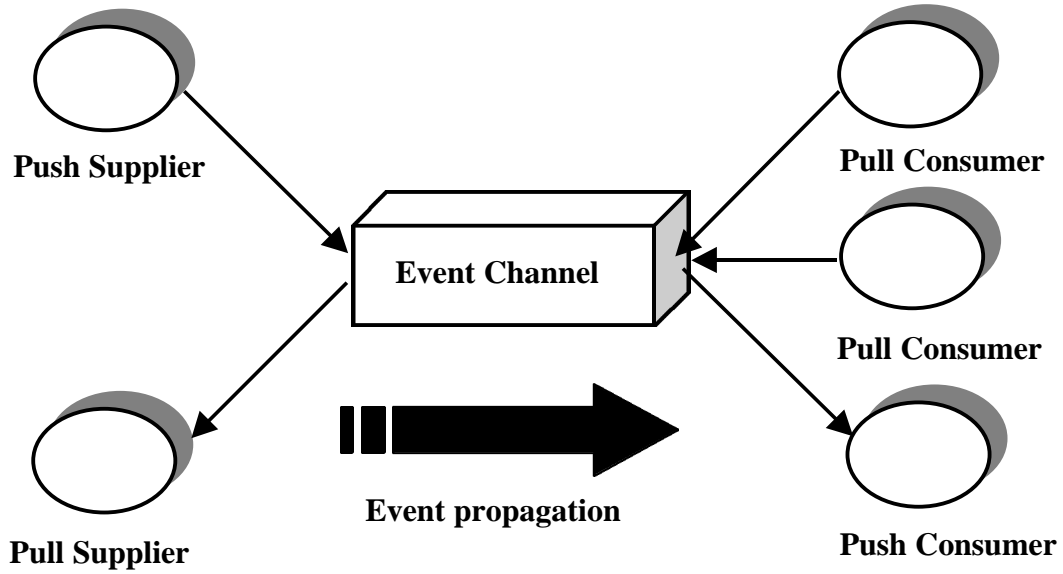


Figure 2-6 Mixing Push and Pull Models on both Suppliers and Consumers

The following table summarizes the role of the Event Channel as a function of the communication model.

Consumer Role	Supplier Role	
	Push	Pull
Push	Notifier	Agent
Pull	Queue	Procurer

Table 2-1 Roles of the Event Channel

Point to Point Events

The CORBA Event Service also provides interfaces that allow a supplier to contact a consumer directly, as shown in Figure 2-7, [OHE:97]. The trouble with point to point communication is that it removes the decoupling between the supplier and the consumer. The supplier must keep track of the consumer and the consumer must keep track of the supplier. It also means that the implementation of a consumer must be able to handle the situation when it cannot keep up with the supplied event data.

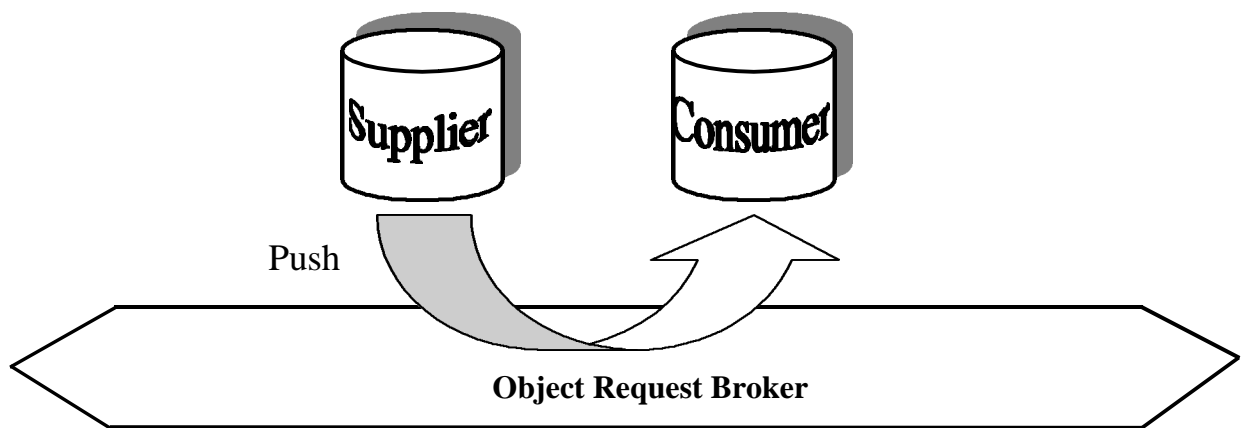


Figure 2-7 Point to Point Communication

In most cases this type of communication is not used as suppliers and consumers are dependant on each other.

Types of Event Communication

The CORBA Event Service maps an event to a set of operation calls [DSSV9:97]. These operations and their sequence are clearly defined for both, the Push and the Pull models, along with the data about an event which can be operation parameters and return values. The data is application specific and can take one of two forms, *typed* and *untyped* (generic).

Untyped Event Communication

In an untyped event communication, events are propagated by a series of generic push or pull operation calls. Event data parameter for the push operation call is a single parameter of type **any**. This allows any IDL defined data type to be passed between suppliers and consumers. For the pull operation there are no parameters, the event data is transmitted in the return type which is also of type **any**. In both, the push and the pull cases supplier and consumer applications must agree on the contents of the **any** parameter. The generic Event Channel does not understand the contents of the data it is passing.

Typed Event Communication

A typed event communication allows applications to describe the contents of the event. The programmer who defines application specific IDL interfaces through which events are propagated, achieves this. Parameters must be input only, no values are returned. In the push model, communication is initiated simply by invoking operations on the interface. The pull model is more complex, as event communication is initiated by invoking operations on an interface that has been specifically constructed for the application. Typing of events is a powerful means of filtering event information. Objects can track the exact events they are interested in.

Interfaces for the CORBA Event Service

The Push Event Interfaces

The push model is the more common way to distribute events in a CORBA based distributed system, especially on wide area networks [OHE:97]. Figure 2-8 shows the four interfaces for the generic (untyped) push style event communication [OMG:95b].

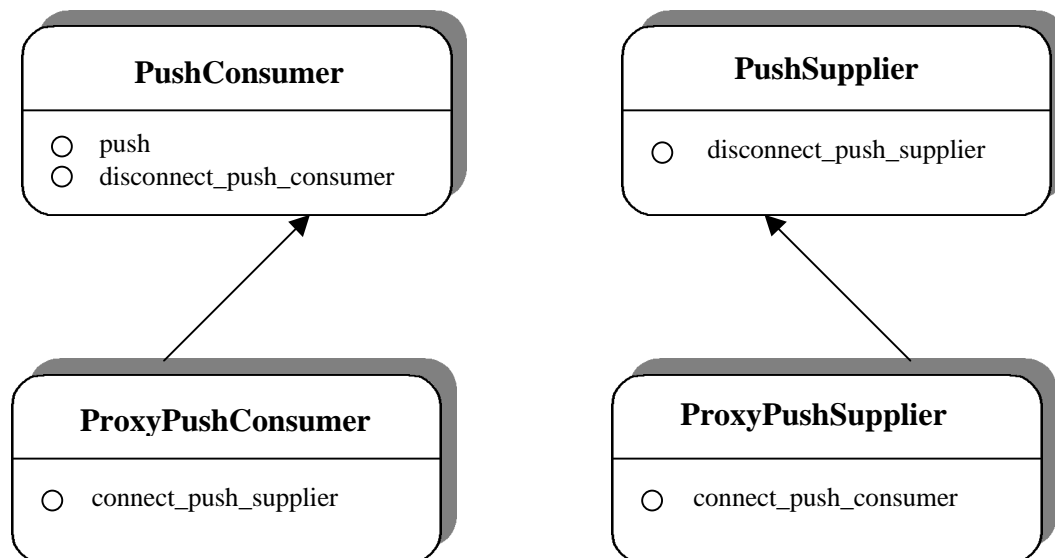


Figure 2-8 The Push Interfaces for the CORBA Event Service

- PushConsumer**
 This contains two methods; *push()* and *disconnect_push_consumer()*. The *push()* method will receive an event with a single parameter of type *any*. An implementation that uses this interface will have to unmarshal the data in the *any* type and then service the event. The *disconnect_push_consumer()* tells the consumer that it will not be receiving anymore events from the Event Channel.
- PushSupplier**
 Contains a single method that stops a push supplier sending events.
- ProxyPushConsumer**
 Contains a single method that connects a push supplier to the Event Channel. This interface simply extends the point-to-point **PushConsumer** interface.

- **ProxyPushSupplier**

Contains a single method that connects a push consumer to the Event Channel.

This interface simply extends the point-to-point **PushSupplier** interface.

The Pull Event Interfaces

The pull model is probably the least used of the two. It is normally used on local area networks since it is a drain on the communication resources [OHE:97]. Figure 2-9 shows the four interfaces for the generic (untyped) pull style event communication [OMG:95b].

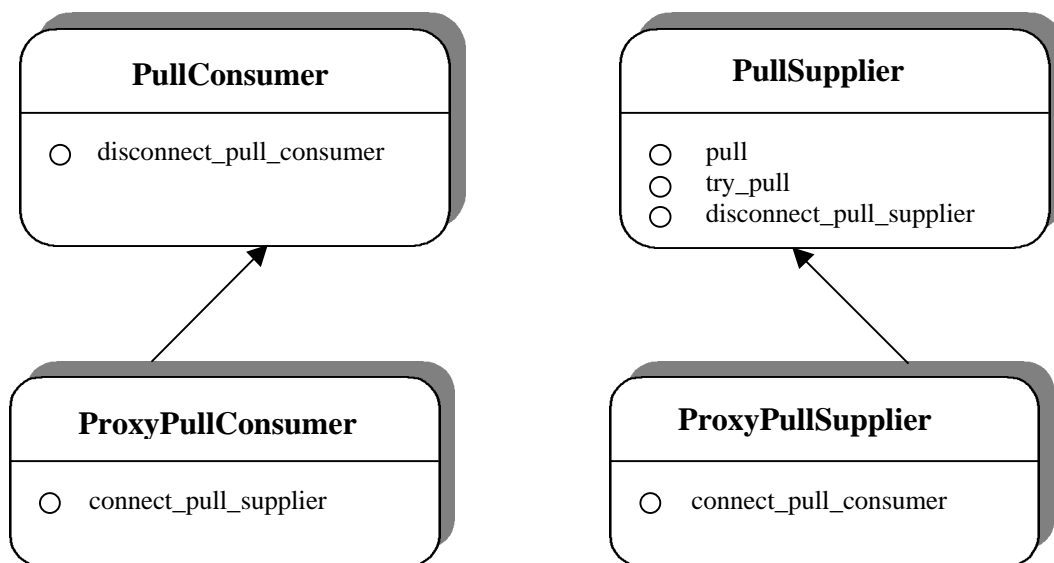


Figure 2-9 The Pull Interfaces for the CORBA Event Service

- **PullConsumer**

Contains a single method that disconnects the pull consumer from the Event Channel.

- **PullSupplier**

This interface consists of three methods; *pull()*, *try_pull()*, and *disconnect_pull_supplier()*. The *pull()* method will block the consumer's execution until a supplier generates an event. The *try_pull()* method allows a consumer to poll the channel for events. The *disconnect_pull_supplier()* removes the supplier from the Event Channel.

- **ProxyPullConsumer**

Contains a single method that connects a pull supplier to the Event Channel. This interface simply extends the point-to-point **PullConsumer** interface.

- **ProxyPullSupplier**

Contains a single method that connects a pull consumer to the Event Channel.

This interface simply extends the point-to-point **PullSupplier** interface.

Typed Interfaces for the Push and Pull Models

There are interfaces that allow push and pull models of event communication to use **Typed** event data. These are very similar to the generic interfaces.

The Administration Interfaces

There are three interfaces that have been specified by the OMG for the purpose of administration within the CORBA Event Service [OMG:95b]. These interfaces are shown in Figure 2-10.

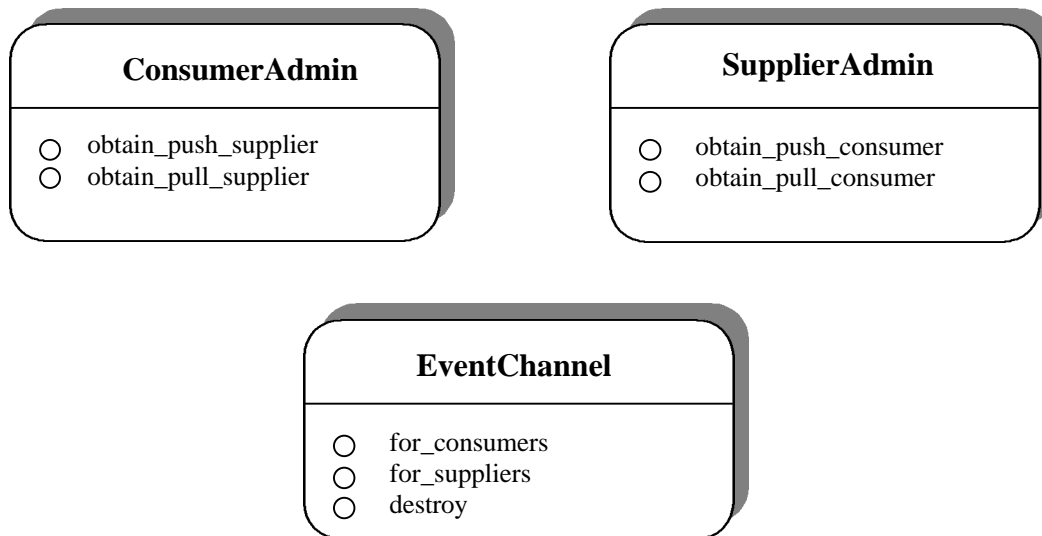


Figure 2-10 The Administration Interfaces for the CORBA Event Service

- **ConsumerAdmin**

This interfaces allows consumers to be connected to the Event Channel.

- **SupplierAdmin**

This interfaces allows suppliers to be connected to the Event Channel.

- **EventChannel**

This interface allows consumers and suppliers to be added to the Event Channel and supports a method for destroying the channel.

The Event Service and Existing CORBA Services

The CORBA Time Service

Determining the order of events in distributed systems is a common requirement, but to achieve this means that the system must have a common clock. This is difficult to achieve due to :

- **Clock Drift** – Physical clocks count time at very slightly different rates.
- **Clock Synchronization** – How to keep clocks on different machines synchronized and how to compensate for clock drifts between synchronization's.
- **Single System Clock Illusion** – How to make all the different system clocks tick at the same time.

The CORBA Time Service provides answers to the above. This service provides the following :

- Determines the order that events occur in a distributed system.
- Obtains the current time along with the error associated with it.
- Generates time based events based on timers and alarms.
- Calculates the interval between events.

It uses the Universal Time Coordinated (UTC) representation of time from the X/Open DCE Time Service. The UTC time in the CORBA specification always refers to the time in the Greenwich time zone.

Time Service consists of five interfaces :

1. Universal Time Object Interface

This represents a UTC time value and its inaccuracy factor.

2. Time Interval Object Interface

Defines operations relevant to time intervals.

3. Time Service Interface

This allows the creation of 'trusted' timestamps associated with events and the specification of the time an event should be fired.

4. Timer Event Handler Interface

This stores the information about a timed event. It includes the time the event is to be fired and the action to be taken.

5. Timer Event Service Interface

This interface is used to create the Timer Event Handler objects and then register and unregister these objects with Event Channels. The event data is passed to the Event Channel, using a push communication model by the Timer Event Handler object at the specified time. The event then gets delivered to all objects that are subscribers to the Event Channel as push consumers.

The CORBA Time Service makes extensive use of the CORBA Event Service push model, in order to deliver events at specific times.

The CORBA Security Service

The CORBA Security Service is built into the ORB and provides security for distributed objects near the high end of the U.S. government security profiles. The specification also allows ORB vendors to provide systems with lower level security. For example, to be able to comply with various export laws. The diagram below shows the architecture for a secure CORBA ORB.

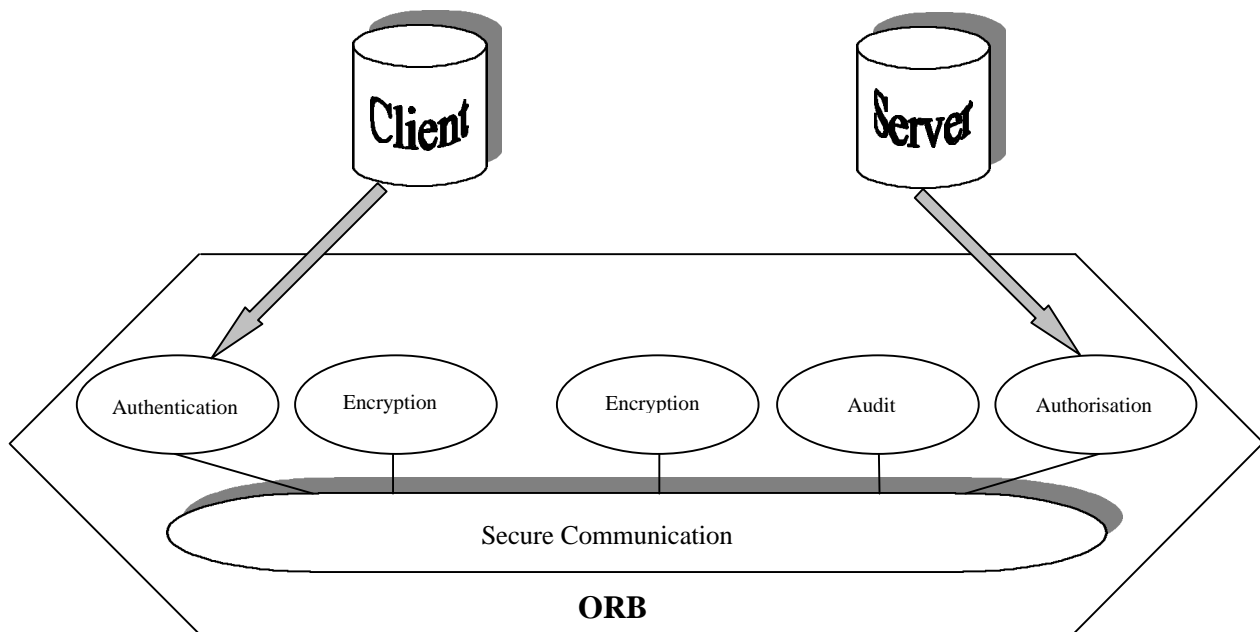


Figure 2-11 The Security Service Architecture model

The majority of distributed objects connected to the ORB will be unaware of the security policy or how it is enforced by the ORB. An aware object will be able to influence the security policy.

The Audit Services, associated with the Server side, are important in terms of events. They allow system managers to monitor ORB events, for example which services or objects are being used. The Audit Services are used to detect attempted attacks on network organizations.

Security aware objects are able to log system and application events. System events include failed object invocations, authentication of principles and updating privileges. Application events are specific to the security needs of the application. For

example, an electronic commerce application that accepts credit card payments could maintain a log for every payment.

The CORBA Security Service lets selected events to be audited by :

- Object or object type.
- Operation.
- Time.
- Principle attributes, such as role or audit ID.
- Success or failure of an operation.

It could be possible, for example to audit all the after-midnight withdraw method invocations on a bank account object.

Audit logs can become very large very quickly. For this reason CORBA defines a set of administrative interfaces that are used to restrict the type of events to be audited. Application events are audited based on audit policies specified by the application developer. System events policies are automatically enforced by the ORB for all types of applications, even for the non security aware applications. Security aware applications use an AuditChannel to record events.

Although the CORBA Security Service does not use elements of the Event Service directly, it does rely on different types of events; system and application for the audit service. The Event Service could be used with the Security Service, were a security aware application uses the Event Channel to decouple the communication between itself and the AuditChannel object.

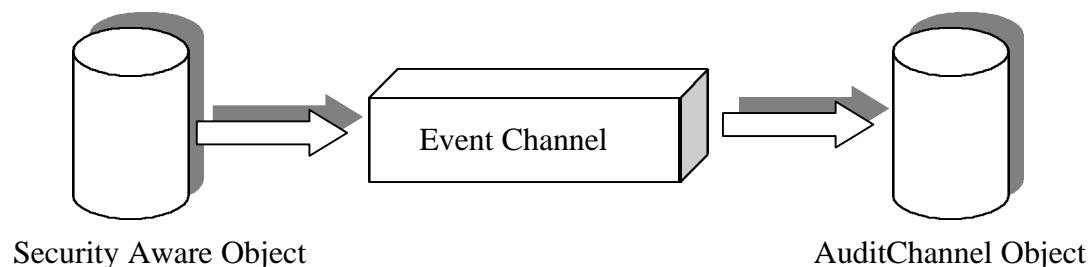


Figure 2-12 Security Objects using the Event Service

Evaluation of the OMG COS Event Service

The COS Event Service model in general is well suited to many applications. It's flexibility allows different implementations to be built, thus avoiding "*one size fits all*" implementations, which in practice do not normally work. This flexibility allows applications to avoid concerns such as timely notifications, multiple registrations and event data persistence and instead rely on Event Channels. The IDL interfaces to the Event Service are simple and event data delivery is fairly straightforward. However, there are several limitations with the OMG COS Event Service specification [DSSV10:97].

- **Over-generalization**

The Event Service is very flexible and can support a wide range of applications. As a consequence the registration of suppliers and consumers is rather complicated due to the multi-step connection process to the Event Channel.

- **Persistence**

The COS Event Service does not specify that the Event Channel must maintain persistence. Therefore, if a host or a process fails, events or the information about the connected consumers and suppliers could be lost.

- **Filtering**

All event data is delivered to every consumer connected to the Event Channel. It is the responsibility of the consumer implementation to filter event data it is not interested in, which can be rather costly and expensive in terms of processing and development times.

- **Correlation**

In some cases, consumers that are registered are unable to execute until they have received events from a particular set of suppliers. The Event Service does not address the event correlation needs of consumers. It is the responsibility of the consumer implementation to perform correlation, which can be very costly. The

consumer workload increases as it has to perform filtering and correlation.

- **Multiple Suppliers**

There are no restrictions on how many suppliers can be connected to an Event Channel. This means more events can be generated which have to be handled by consumers, thus increasing the load on the network.

- **Consumer an Network Load**

Since all events get delivered to all consumers that are connected to the Event Channel, the load on the network increases further. Also the consumer workload will increase as it has to perform the event filtering and correlation. This can cause problems if there are many consumers connected to the Event Channel or if they are run on low powered machines.

- **Lack of Type Safety**

The untyped Event Service passes event data using the CORBA ‘any’ type. This means that the consumer application has to extract the information and convert it to the required type before it is used. However, the CORBA ‘any’ type is error-prone and the consumer has to perform additional error checking.

- **Federation**

In large scale distributed systems, it must be possible for event servers in one logical region (i.e. LAN) to contact event servers in another logical region for forwarding events. The current CORBA Event Service has no mechanism for federating event servers.

- **Quality of Service**

The OMG does not specify the Quality of Service (QoS) that Event Channels must implement. This means that different vendor implementations have different levels of QoS. This can range from “at-most-once” (best effort) semantics to “exactly-once” (guaranteed) semantics. The type of QoS that will be implemented by the ORB vendor will depend on a trade off between performance and duplication or non-delivery of data. For example, a high performance Event

Channel will only implement best-effort semantics, whereas an Event Channel that is concerned with guaranteed delivery will perform slower.

- **Security**

The OMG does not specify any security policies for the Event Service. In many distributed systems, some form of access control is required so that both, supplier and consumer applications are secure. Suppliers can only generate events they have authorization for, and consumers can only receive event data they have been authorized for.

These shortcomings are of particular interest in many distributed application domains such as avionics, telecommunications, process control and distributed interactive simulations. The CORBA Event Service is unsuitable for these types of domains. This has caused many companies to add extra non-CORBA compliant functionality to the Event Service. This leads to a non-standard development of software, which is one of the issues that the OMG is trying to solve with the CORBA specification. The OMG has issued a request for a Notification Service and with the introduction of the CORBA 3.0 specification the problems that plague the Event Service should be non-existent in the future. The next two sections describe the Notification Service and the CORBA 3.0 specification in relation to the Event Service.

The Notification Service

One of the very fundamental requirements of large scale distributed systems is event management. The OMG proposed the Event Service specification, which has been implemented by a number of CORBA vendors. However, there are a lot of drawbacks which makes the Event Service unsuitable for many distributed applications, in particular telecommunication network applications. For this reason the OMG has made a Request for Proposal (RFP) for a Notification Service [OMG:96f]. To avoid confusion with the Event Service, the data managed by the Notification Service is referred to as “notifications”. This section describes the proposal of the Notification Service and how it fits in with the existing Event Service.

The OMG Event Service provides a basic level of support for event style communications in CORBA-based applications. Since many distributed system architectures are for large scale systems with good scalability, the amount of event traffic becomes potentially very large. The basic event filtering mechanisms in the Event Service for typed events are exceeded. It is possible for CORBA vendors to provide their own more sophisticated means of event filtering. However, this has the adverse effect of losing interoperability between various vendor solutions. The main purpose of the Notification Service is to provide standard interfaces for supporting filtering of events.

Relationship to Existing CORBA Services

The Notification Service will be based on the OMG Event Service, it will basically be an extension providing solutions to the various drawbacks. The relationship between the Notification Service and the Event Service has yet to be decided. But they will probably co-exist with the Notification Service being used in large scale distributed systems that require extensive filtering of event data.

The Notification Service will use the CORBA Security Service for controlling access of operations that subscribe to various notifications and allow consumers to access various notifications.

General Requirements

The Notification Service is very similar to the Event Service. Notifications are sent by notification producers and delivered to notification consumers. The producers and consumers are decoupled to allow scalable configurations. The communication styles supported are both push and pull for both, typed and untyped events, as in the Event Service. The Notification Service will also preserve all data, with the identity of the original producer not being lost.

Notification Delivery

The producer has to discover the Notification Server, then start publishing the notifications. The consumer will also discover the server, subscribe to the notifications of interest and then receive them.

Notification Subscription

This is the filtering mechanism, consumers register their interest with the server. Consumers can terminate their interest in notifications at any time. The filtering will be flexible and can therefore work with arbitrary parameters.

Management of Producers and Consumers

As with the Event Service, producers and consumers can register with the server at any time. When the producer registers, it informs the server the type of notifications it will be generating. Consumers, when registering, tells the server which notifications are of interest, along with any extra filtering criteria.

Security

The Notification Service will be used alongside the OMG Security Service. Producers can only deliver authorized notifications to the server and consumers can only receive notifications that have been authorized.

Federation of the Notification Server

Notification Servers in different organization networks will be aware of each other. This will allow producers in one organization network to send notifications to consumers in another organization network.

Qualities of Service

Applications require different Quality of Service requirements. The Notification Service will provide various levels of QoS to meet the varied application needs. The levels of QoS are identical to that being proposed by the Messaging Service [MOM:98]. The following is the QoS that the Notification Service will support.

- **Assured Notification Delivery**

Notifications are guaranteed to be delivered to all consumers. If consumers are unavailable for some reason then notifications will be delivered as soon as the consumers become available again.

- **Notification Priority and Assured Delivery Ordering**

Notifications will be delivered in a FIFO order within a priority. Higher priority notifications being delivered first to consumers.

- **Aging of Notifications**

Consumers can be unavailable for a number of different reasons. It will be possible to configure the Notification Service so that it will only attempt to deliver notifications to unavailable consumers for a period of time. Once this time has expired notifications are discarded.

- **At-Least-Once Guaranteed Delivery**

With this configuration, the Notification Service will retain a notification until it has been delivered to at least one consumer.

CORBA 3.0 – The Next Generation

Currently all CORBA vendors provide implementations that are compliant with the CORBA 2.0 specification. The CORBA 3.0 specification includes many new features including; multiple interfaces, server-side portable frameworks, pass-by-value, mobile agents, business objects and messaging.

The messaging feature is of interest with regard to events in distributed systems. This section introduces the messaging service and its benefits.

CORBA 3.0 Messaging

At the basic level, CORBA is a synchronous request-reply protocol where the client blocks until the operation has been completed by the server and the result returned. If you do not want the client to block or to put the calling invocation in its own thread, CORBA provides three solutions :

1. Oneway methods.
2. DII deferred synchronous mode.
3. CORBA Event Service.

These three solutions do not provide the same solution as the Message Oriented Middleware (MOM) [MOM:98]. But with the CORBA Messaging Service proposal [OMG:96e], it will allow ORB's to provide MOM solutions using the CORBA model.

Message Oriented Middleware – An Introduction

MOM allows messages to be exchanged between clients and servers asynchronously by using message queue. A client will create a message and places it in the queue. A server at some other period in time after the message has been placed in the queue will take the message and act upon it. A response will only be sent back to the client if it requires one.

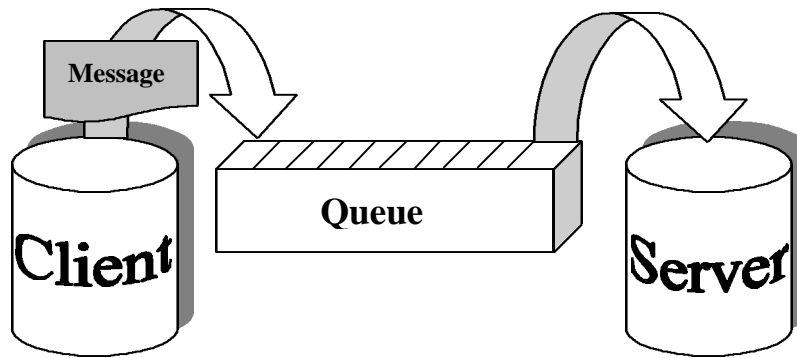


Figure 2-13 General Architecture of the MOM

This allows clients and servers to function at their own designated times and speeds without necessarily being simultaneously active. The message queue is very flexible it can be used with one-to-many or many-to-one relationships. The server can take messages from the queue either on a FIFO basis or according to some priority or load balancing scheme. Servers can also use message filters to disregard the messages that they are not interested in. Queues can either be persistent (logged to disk) or non-persistent (in memory). The type of queue required determines on the Quality of Service. Some MOM's provide some level of fault-tolerance either in the form of persistent queues or transactional processing in a two-phase synchronisation protocol. MOM's may even re-route messages to other queues if a queue fails or there are network problems.

Some of the benefits that MOM brings to CORBA are :

1. Clients do not have to block to make requests. This makes the client program much easier since it does not have to be multithreaded.
2. Clients and Servers can run at different times. For requests to be completed successfully both, the client and the server do not have to be available at the same time. For example, if a server has failed and a client sends a message to that failed server, the message will remain in the queue until the server becomes available when it can retrieve and process the message accordingly.
3. It supports nomadic clients. Disconnected from the server, clients will be able to accumulate outgoing message requests in queues until the connection with the server is established.

4. Servers can determine when to retrieve messages. A server can take messages from the queue several hours after the message was added by the client.

MOM style communication will make CORBA more flexible and time tolerant. With this, next generation CORBA MOM will be integrated with the Event Service to provide very powerful publish-and-subscribe systems.

Summary

The CORBA Event Service allows applications to communicate with each other in an asynchronous manner by providing a decoupling object, an Event Channel. A supplier application is connected to one end of the Event Channel and generates events. A consumer application is connected to the other end of the Event Channel, it receives events and processes the data accordingly. The Event Channel broadcasts the event information to consumers, all consumers see the same event whether they are interested in it or not. The CORBA Event Service is suitable for many application domains, but due to the drawbacks it makes it unsuitable for many other application domains, such as telecommunications and real-time systems. The OMG is currently proposing a Notification Service which will address some of the issues such as; filtering, event priority, persistent Event Channel, but it does not go all the way to solving the problems. The next generation of CORBA will include a Messaging Service which is based on the Message Oriented Middleware technology, and this will address the remaining issues.

3

Academic and Commercial Event Models

Many CORBA vendor companies have developed Event Services, compliant with the OMG specification. Some have even added their own functionality to overcome the drawbacks depending on the application domain the Event Service is to be used in. Events are still a major research interest in distributed systems in both CORBA and non-CORBA communities. This chapter describes a selection of commercial and academic event models that are CORBA and non-CORBA based, and looks at their advantages and disadvantages.

CORBA Event Models

IONA Messaging Services

IONA Technologies provides two different types of message service products; OrbixEvents and OrbixTalk.

OrbixEvents

OrbixEvents [IONA:97] is a C++ implementation of the OMG CORBA Event Service. It is the only commercial available Event Service that supports both, untyped and typed events for the push and pull communication models. OrbixEvents can use

either IIOP or the proprietary Orbix protocol.

Since OrbixEvents can use IIOP this gives it the advantage that it can interoperate with any other IIOP compliant ORB. One of its drawbacks is that all events must pass through an Event Server. This forms a bottleneck for the system, which can cause scalability problems for large scale distributed systems. Also there is no multicasting of events, so event data has to be sent individually to each consumer. This is a waste of bandwidth if the consumer is not interested in the event.

OrbixTalk

OrbixTalk [IONA:96] is the second messaging service product from IONA Technologies. This product is used for distributing IDL-based operations over UDP using either a simple or reliable multicasting. It has an implementation of the CORBA Event Service. But since it is based on UDP it cannot interoperate with any other ORB system. However, OrbixTalk is ideal in systems that have many consumers and suppliers, since with UDP there is no need to maintain the connection between each consumer and supplier.

The event data does not have to pass through a central Event Server because OrbixTalk supports multicasting. Thus the bottleneck is removed and OrbixTalk applications scale much better for large scale distributed systems. It also provides a "MessageStore" object that can be used for persistence and guaranteed message delivery.

jEVENTS

This is a Java implementation of the CORBA Event Service for untyped messages [OUT:97], produced by a company called Outback Resource Group Inc. jEVENTS supports both push and pull style communication between suppliers and consumers. It is multi-threaded and takes advantage of the strengths of Java. It is also IIOP compliant and may be used with any ORB that supports IIOP.

VisiBroker Event Service

The VisiBroker Event Service is available in both, C++ and Java versions from Inprise Corporation. Both versions are compliant with the OMG CORBA Event Service implementing untyped events for push and pull communication models. When used with VisiBroker ORB and its Smart Agent architecture that is vendor specific, it becomes a highly available self-recovering service. If an Event Service fails, the Smart Agent can automatically reroute the request to a node that can activate a new Event Service.

DAIS Multicast Event Service

ICL have developed a multicast Event Service called DAIS [ICL:98]. A multicast service was developed due to the requirements from a customer that needed to communicate sixty messages per second, each being a few hundred bytes in size, to over eight hundred consumers. With these requirements, a standard CORBA Event Channel would have to produce nearly half a million individual messages per second, clearly unfeasible for a distributed system.

The DAIS Event Service is based on the existing OMG CORBA Event Service specification, but has added extra non OMG multicast IDL interfaces.

To minimize the amount of network traffic between supplier and consumer applications, messages are collected into packets and sent in a single UDP packet. Since UDP does not handle message fragmentation, the message fragmentation is applied before being delivered. UDP is also unreliable, it is therefore the responsibility of the consumer application to keep track of lost messages. When a lost packet has been noticed, the consumer must make a request to the supplier for re-transmission.

DAIS uses UDP which means that it does not support IIOP, and therefore cannot interoperate with other ORB's.

TAO – Real-time Event Service

TAO is a real time ORB that has been developed by the Computer Science department at Washington University. One of the components of this ORB is a real-

time Event Service [HLS:97] that is an extension of the CORBA Event Service. It provides source based and typed based filtering, event correlation's, and real-time dispatching.

The architecture of this real-time Event Service compared to the CORBA Event Service is described below :

- **Event Channel**

This plays the same role as the CORBA Event Channel.

- **ProxyPushSupplier Interface**

This has been extended so consumers can register their execution dependencies with the channel.

- **ProxyPushConsumer Interface**

This has been extended so suppliers can specify the type of events that they generate. This allows channel subscription and filtering modules to build data structures that allow efficient run-time lookups of the subscribed consumers.

- **Subscription and Filtering**

The CORBA Event Service defines Event Channels as broadcasters. This means that generated event data gets delivered to all consumers connected to the channel whether they are interested in the event or not. This can be very wasteful when there are many suppliers and consumers connected to a channel. The TAO's real-time Event Service extends the CORBA Event Service interfaces to allow consumers to subscribe for a particular subset of events. The channel then uses the subscriptions for filtering events and only delivers the events to the interested consumers.

- **Priority Timers Proxy**

Consumers can register with the Event Channel for a specified amount of time.

The priority timer proxy cooperates with the run-time scheduler to ensure that a

consumer application has a chance to process the data before it times out, and disconnects itself from the channel.

- **Event Correlation**

Some consumers require event data from a number of different suppliers before they can execute. Together with the filtering, the correlation mechanism allows consumers to specify logical ‘AND’ and ‘OR’ event dependencies. When a consumer dependency is met, the real-time service delivers the event data.

- **Dispatching**

This determines when events should be delivered to consumers. It cooperates with TAO’s scheduling service to ensure consumer execution within their deadlines.

TAO’s real-time Event Service arguments the CORBA Event Service model, by providing solutions to its drawbacks. The push driven model is commonly used in many real-time environments due to the efficient and predictable execution of operations. In contrast, using the pull model would mean if event data is not available to be pulled, the calling consumer would block awaiting a result. In order for a consumer to block, the consumer must allocate additional threads to allow the application to continue while waiting for a response. Adding extra threads to a system has many negative effects such as, increased switching overhead, complex synchronization, and complex real-time thread scheduling policies. For these reasons TAO’s real time Event Service is only concerned with the push style of communication. The real-time Event Service is written in C++ and deployed to McDonnell Douglas in St. Louis, where it is being used to develop operation flight programs for the next generation avionics systems.

COBEA – COrba Based Event Architecture

COBEA [CMJB:98] is an Event Service designed and developed by the University of Cambridge Computer Laboratory. COBEA is a scalable system based on the publish-register-notify model. This system is an extension of the CORBA Event Service that supports parameterized filtering, fault-tolerance, access control and composite events, all of which are not in the OMG Event Service specification.

The architecture consists of sink and source interfaces, a mediator and a composite event server. The purpose of the sink interface is to receive events from suppliers and support event notification. The event source interface is used for event registration and de-registration. The mediator provides the decoupling that is required between supplier and consumer applications. Consumer applications may require many events from many suppliers, and for these events to arrive in a specific order, for the consumer to carry out the action successfully. With the composite event server, the COBEA system provides this.

Events can occur before a consumer has registered their interest, or a consumer cannot digest all the events being supplied. COBEA provides event buffers for suppliers or mediators. A time-to-live (TTL) parameter is associated with each event to ensure it does not get discarded before being delivered. Priority can also be specified with each event. Event queues are maintained according to priority. Events are delivered in a FIFO order within the priority queues. Events can be delivered in 'fast' mode, which does not guarantee the delivery of an event, or a reliable mode can be used. In the COBEA system, consumer applications must supply their role name when registering. This is because many applications require that events are only delivered to authorized consumers. The role name is validated by the server to ensure the consumer has the access rights for a particular event.

COBEA provides a powerful extension to the CORBA Event Service, so that it can be used in application domains such as telecommunications, where a single source can generate hundreds of events per second. It focuses more on the push model which is the core of event communication. Continual work is being carried out on COBEA, which will include incorporating security measures and implementing support for reliable event delivery.

JacORB

JacORB [GB:98] is an Object Request Broker that is a partial implementation of the OMG CORBA standard written in Java. It was developed by a research assistant at the University of Berlin. It contains a prototype implementation of the CORBA Event

Service. The Event Service supports both, push and pull style communication for untyped events.

ORBacus

ORBacus, formerly known as OmniBroker has been developed by a company called Object-Orientated Concepts Inc [OOC:98]. ORBacus is a full featured ORB for both, C++ and Java. It is compliant with the CORBA 2 specification and contains an implementation of the CORBA Event Service. The Event Service supports both, push and pull style communication for untyped events.

Non-CORBA Event Models

Elvin

Elvin [Elvin:97] is a notification system under development at the Distributed Systems Technology Centre, University of Queensland. Elvin has been ported to a number of different platforms including; Solaris, HP-UX, OS/2 and AIX. The current release version is 3.0, which is implemented in the C programming language using the TCP/IP communication protocol.

Elvin is a publish/subscribe notification service. The producer (supplier) publishes a notification through the server. The notification is then placed on a queue and evaluated by a subscription engine that consists of a database. The notification is then delivered to all consumers that have matching subscriptions. Consumers subscribe by sending a subscription to the subscription queue which causes the subscription engine to be updated. This avoids the use of a channel to select event types, which creates a dependency between the consumer and producer of notifications and vice-versa.

Fundamental to the Elvin design is the mechanism of reverse subscription, called quench expressions. The purpose is to reduce network traffic so that notifications only get delivered to consumers that are interested in the information. A quench expression is where the producer obtains an active subscription expression in the server. This allows the fan-out to be reduced by stopping producers sending notifications for which there are no consumers.

Elvin version 4.0 is currently under development with the main focus on, federation, extended language bindings and greater platform support. For reliability and security, there have been a few minor architecture changes.

InfoBus

The InfoBus communication model concept was created at Lotus, and was jointly developed by Lotus, Sun Microsystems and Oracle Corporation. The architecture is very similar to the CORBA Event Service, a producer (supplier) application communicates asynchronously with a consumer application via an object bus. In terms of CORBA the object bus is the Event Channel. Producer and consumer applications have to be built using Java Beans and must be running on the same Java Virtual Machine (JVM) in order to exchange data. An individual component can act as both a producer and consumer. There are three types of Java Bean classifications in the InfoBus system :

- **Data Producers**

These are components that deliver data to consumer components at their request.

- **Data Consumers**

These are components that are interested in receiving new data.

- **Data Controllers**

These are components that regulate or redirect the flow of data between producers and consumers.

The data that can be exchanged between producer and consumer can be either simple items such as, numbers and strings, or more complex items such as, lists, arrays and tables. Any Java component can connect to the InfoBus. Once connected, all InfoBus notifications arrive via the standard Java event mechanism. The role of data producers is to make new data available to the InfoBus as it becomes ready. The data consumers will take the data as they require it. The InfoBus system maintains a membership view of the connected producers and consumers. When producers/consumers join or leave the InfoBus their view is changed accordingly. The system also defines different security policies that control whether a member can join or register with the InfoBus instance.

While the InfoBus is primarily intended for communication among Java components in the same Java Virtual Machine, it is possible for a component that supports either

IOP or Remote Method Invocation (RMI) to obtain data that is published on the InfoBus. For example, the Internet Calendar Access Protocol which supports IOP could obtain a personal calendar from the data repository and publish the item on the InfoBus. Or a component which uses RMI could communicate with another component in a different JVM and then publish the data on the InfoBus.

iBus

This is a commercial product from SoftWired, a Swiss software company located in Zurich. iBus [iBus:97] is implemented in Java, the architecture model is similar to the CORBA Event Service in that suppliers communicate with consumer applications via a channel. iBus supports both asynchronous push and synchronous pull style communication, the channel is named by Uniform Resource Locators. A major feature of iBus is its protocol framework. This provides different levels of Quality of Service such as reliable IP multicast using negative acknowledgements, unreliable IP multicast, TCP communication, message encryption, failure notification and virtual synchrony.

iBus is intended for use by mainly Java applications whereas the OMG Event Service supports interactions among applications written in different languages. iBus is much more lightweight in that it does not require any IDL compiler or platform specific libraries. Also a user defined object can be transmitted between supplier and consumer, the CORBA Event Service is much more restrictive. iBus can be used with CORBA applications by using an IOP bridge.

ECO – Events, Constraints and Objects Model

ECO is an event based object model for distributed programming, developed by the Distributed Systems Group at Trinity College.

The ECO model [SCT:95] is based on Events, Constraints and Objects to simplify the more difficult issues involved in programming distributed systems. Such issues include; timing, synchronization and communication. In this model, objects communicate with each other, possibly over multiple nodes. The sending object announces events and is not concerned with the identity or destination of the receiving

object. The receiving object registers their interest in events, without concern for the sending objects. The decoupling is achieved with event handlers that are invoked when an event is announced. Constraints are named conditions which control the handling and delivery of events. Concurrency, synchronization and timing properties can be expressed using constraints which can be associated with objects and events.

The ECO system was implemented for use in a single address space. A library is used to maintain information about the objects, events and constraints. This library is then linked in with the application code.

Summary

Many commercial and academic event models have been developed for both CORBA and non-CORBA systems. Most of the CORBA commercial implementations, with the exception of DIAS, do not add their own functionality to overcome some of the drawbacks. The problem of adding non-compliant OMG functionality is that the service is not interoperable with other ORB's. Academic CORBA implementations, such as TAO and COBEA have provided many interesting solutions to the CORBA Event Service, so they can be used in telecommunication and real-time domains. The non-CORBA models are based on similar principles as the CORBA Event Service and the majority can be used with CORBA applications by using an IIOP bridge. It seems that currently there is not one event model that fits all application domains, the best way to select a suitable event model is to look at the domain and find the model that matches best.

4

Requirements for TCDEvents

This chapter describes the requirements for TCDEvents, a Java implementation of the CORBA Event Service.

System Model

The overall system architecture of is shown in Figure 4-1. It consists of four elements that are all located at different network hosts.

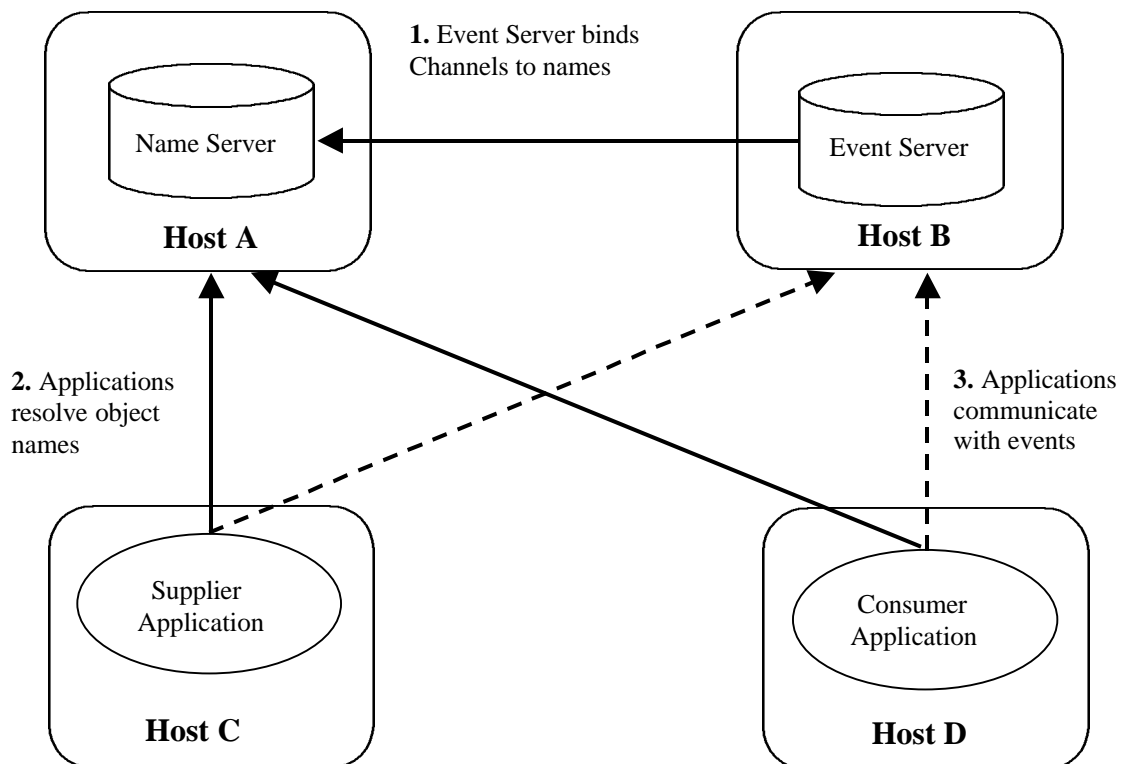


Figure 4-1 System model for TCDEvents

The Event Server registers every name of the Event Channel with the Name Server. This allows supplier and consumer applications to obtain the reference of the channel they require by contacting the Naming Service. Once applications have a handle on the required Event Channel then they are able to generate or receive events.

The CORBA Event Service consists of three ORB objects, suppliers, consumers and channels. The event data passed from suppliers to consumers are not objects because the CORBA distributed object model does not support passing objects by value. The interfaces for these objects and the two types of communication models that they support (push and pull styles) have been defined by the OMG using the Interface Definition Language (IDL).

Functional Requirements

This section describes the functional requirements for the Event Service.

General

The Event Service should support the CORBA Internet Inter-ORB Protocol (IIOP), so allowing the software engineer/system integrator to develop event applications using any IIOP compatible ORB.

The Event Service needs to have minimal configuration requirements beyond those of the ORB that is used for developing applications.

Locating the Event Channel

Developing event applications will use an implementation of the Naming Service to obtain a reference to the Event Channel. The Naming Service provides the CORBA compliant way for obtaining references to objects.

The Event Channel

There are three communication styles with an Event Channel :

1. Event Channel using the Push Style

The supplier pushes event data to the Event Channel. The Event Channel then pushes event data to the consumers.

2. Event Channel using the Pull Style

The consumer pulls event data from the Event Channel. The Event Channel pulls event data from the supplier and the consumer receives the event data.

3. Event Channel using both Push and Pull Styles

The Event Channel can communicate with the supplier using one style (i.e. the pull style), and communicate with the consumer using a different style (i.e. the push style).

The Event Channel object is created when the server application is started, there will be no supplier or consumer objects connected to it. Each channel that is created within the server will have a unique identifier. Using the Naming Service, the user will be able to obtain the reference to the desired channel.

Every Event Channel has an associated buffer that limits the number of events in transit. The size of the buffer will have a default size, but the user will be able to change the size by using a command line argument when registering the server in the Implementation Repository. It will also be possible to change the buffer size by invoking a method from the Event Channel object Interface.

The *EventChannel* Interface defines three operations. One for being able to add a consumer object, the second for adding supplier objects and the third for destroying itself. When an Event Channel gets destroyed, all suppliers and consumer objects that were connected, also get destroyed.

The Channel Manager

The Channel Manager object is part of the server, and allows the developers of applications to manage the Event Channels within the server. The object will support an IDL interface that will allow creating, finding and destroying of Event Channels.

A reference to the Channel Manager object will be obtained by resolving its name with the Naming Service.

The Consumer

A consumer object is created by calling the *for_consumers()* operation from the **EventChannel** Interface. This returns a reference to the **ConsumerAdmin** Interface. The **ConsumerAdmin** Interface contains two operations that both return references to the Event Channel supplier proxy. The consumer administration reference then calls one of these operations, depending if it is to be connected as either a push consumer or a pull consumer. The consumer then finally registers with the Event Channel by calling an operation from the proxy supplier interface.

It is the responsibility of the consumer application to obtain the reference of the required Event Channel by using the Naming Service.

The Supplier

A supplier object is created by calling the *for_suppliers()* operation from the **EventChannel** Interface. This returns a reference to the **SupplierAdmin** Interface. The **SupplierAdmin** Interface contains two operations that both return references to the Event Channel consumer proxy. The supplier administration reference then calls one of these operations, depending if it is to be connected to the Event Channel as either a push supplier or a pull supplier. The supplier then finally registers with the Event Channel by calling the operation from the proxy consumer interface.

It is the responsibility of the supplier application to obtain the reference of the required Event Channel by using the Naming Service.

Event Delivery

When a supplier generates an event, that event gets delivered to all the consumers that are connected to the Event Channel regardless if the event is of interest to the consumer or not. The Event Channel will support the minimal requirements for Quality of Service. It will only support “best-effort” delivery of events. Application domains that require event-style communication to be more reliable than “best-effort” semantics, for example “exactly-once” semantics, will have to make use of the CORBA Notification Service. The order at which events are delivered to the consumers will not be addressed, as this can be solved with the forthcoming CORBA Message Service.

The Event Server

The Event Server will be registered with the Implementation Repository, and will be available at each host where an Event Channel is required. It will contain the Event Channel object and the Channel Manager object.

The Event Server will interact with the Naming Service, to register the channel name with the Name Server. If a channel name has already been registered with the Name

Server then by default the channel will not be registered and the server will except the failure. It will be possible for the server to force the Name Server to replace the current binding for the new channel name. This will be achieved by specifying a switch command at the command line when registering the server with the Implementation Repository.

Additional Requirements

One of the problems with the Event Service is the multiple steps required to connect the supplier/consumer to the Event Channel. All of these steps are the same for every supplier/consumer application. It would be a good idea to remove this burden of coding from the software developer/system integrator by having a Java GUI tool that generates the code for these steps. The steps that would be generated are :

- Obtain a reference to an Event Channel object. This would be achieved by calling a member function, that would use the Naming Service to obtain the reference to the desired channel.
- Get the supplier/consumer administration object reference, using the reference from the Event Channel.
- Get the proxy push/pull object reference for the supplier/consumer using the administration object reference.
- Connect the push/pull supplier/consumer to the Event Channel.
- Push/Pull the events to/from the Event Channel.

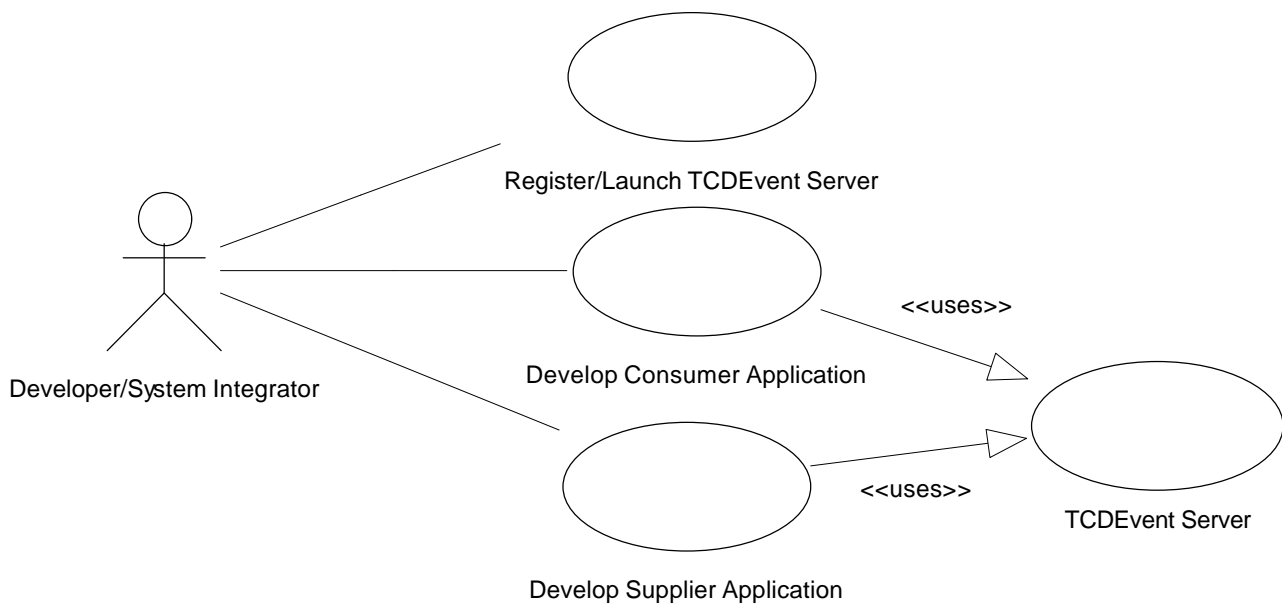
Use Case Models

This section shows a few of the most common use case models for TCDEvents. The first model shows a case for the server, the second for the developer, the third for the supplier and consumer and finally the fourth is a use case for the Event Channel.

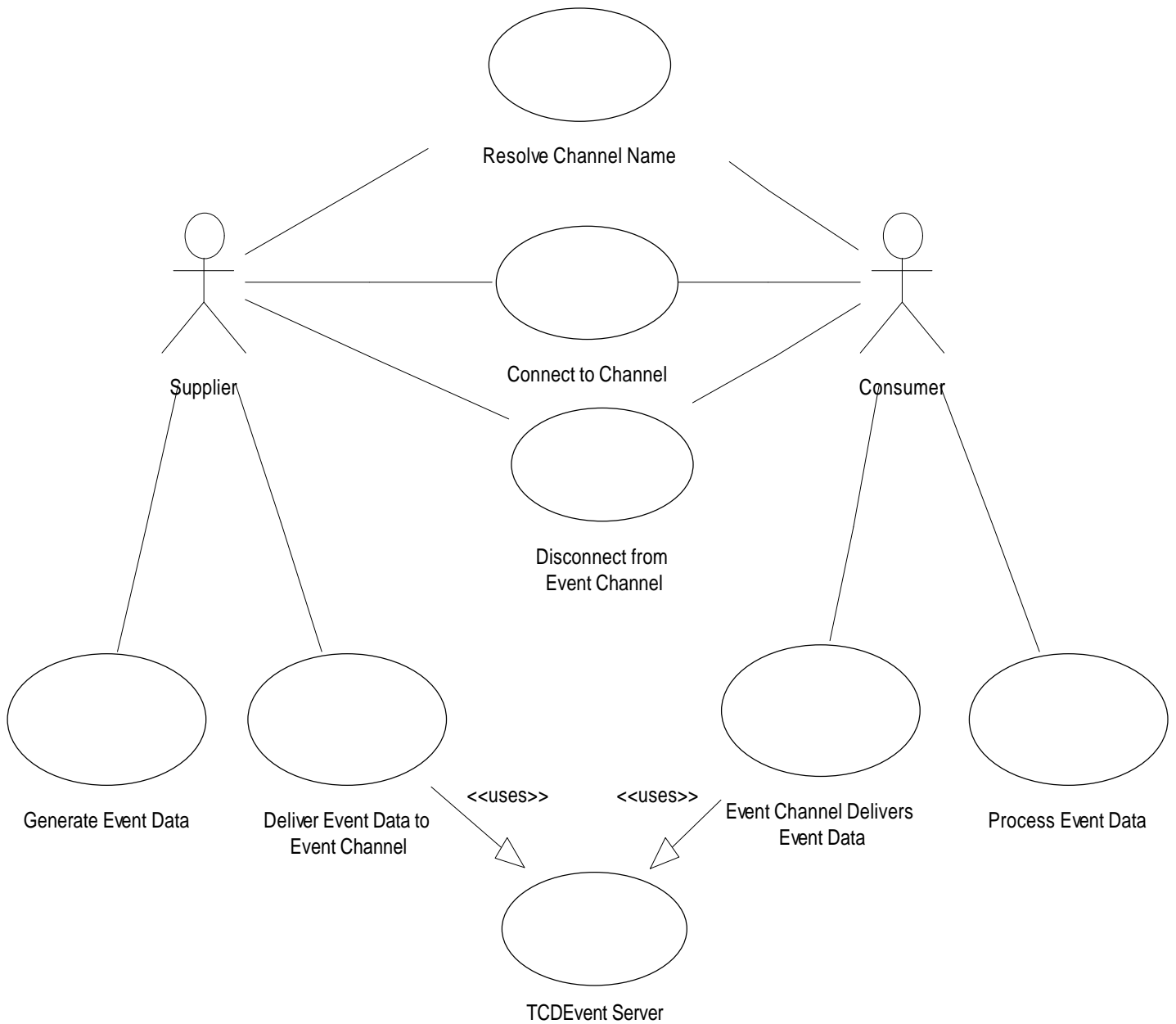
- **The TCDEvents Server**



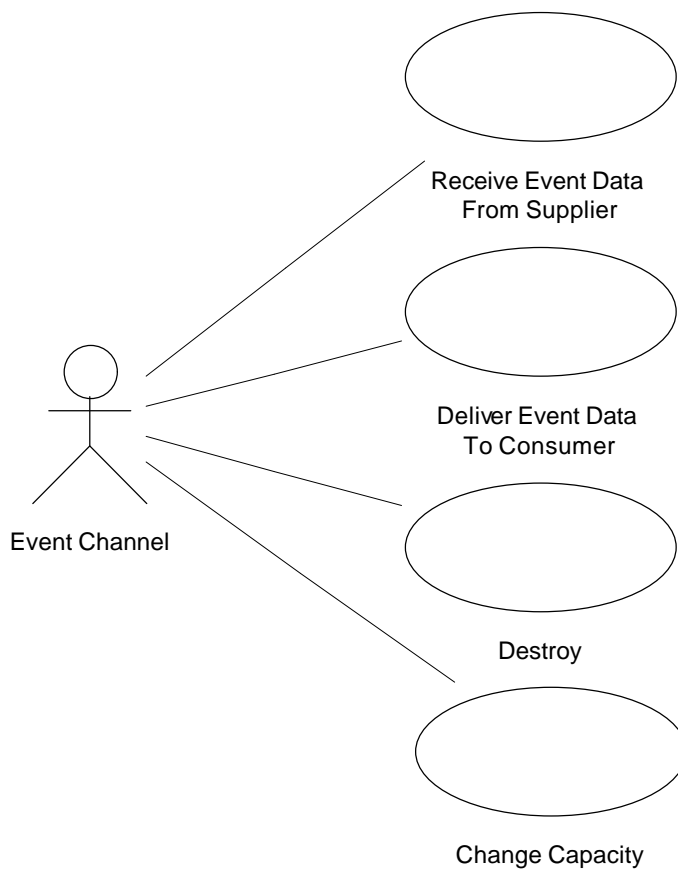
- **Developer or System Integrator**



- **Supplier and Consumer**



- **Event Channel**



Summary

TCDEvents will be written in the Java programming language taking advantage of its strengths. These strengths include exception handling, strong type checking, and the ability to run on multiple platforms. It will be independent of any vendor specific ORB's and will be able to "plug and play" on any operating system that supports the Java Virtual Machine and CORBA. One of the major requirements is that it is easy to use and removes much of the burden from the software developer by providing an Event Server and a GUI wizard.

5

Design and Implementation of TCDEvents

This chapter describes in detail the design of a CORBA compliant Event Service, TCDEvents, as outlined by the requirements described in Chapter 4. The design uses an object orientated strategy and makes extensive use of the Unified Modeling Language (UML).

The design is described independent of any object oriented programming language, even though the Event Service will be implemented purely in the Java programming language. The Event Service will take full advantage of the strengths of the Java programming language such as exception handling, strong type checking and the ability to run on multiple platforms. It will also implement features that minimize resource utilization, including thread pool management, and the ability to limit the number of events in transit. TCDEvents will be independent of any vendor specific ORB's as it will fully support the CORBA Internet Inter-ORB Protocol (IIOP).

The Event Service will be designed, so that it is easy to use and removes as much as possible the programming burden away from the software developer/system integrator. The main aim is that TCDEvents can be used within a distributed system with minimal amount of effort.

Detail Design

This section describes in detail the design of a CORBA compliant Event Service, independent of the implementation programming language. The design uses the Unified Modeling Language (UML) to model the system. UML is a standard language for visualizing, specifying, constructing and documenting elements of a software system. Use Cases (detailed in Chapter 4), Interaction Diagrams, Class Diagrams and Package Diagrams were developed for this project.

To create a CORBA event application, the software developer/system integrator must implement consumer and supplier applications using an Object Request Broker (ORB). These consumers and suppliers communicate through the server application that will be provided.

Event Server

The Event Service will implement a server application that supports untyped events. The software developer/system integrator will register the server application with the Implementation Repository. The server will automatically bind the Event Channel name with a CORBA compliant Naming Service.

The server will be able to contain one or more Event Channels. There will be no limit to the number of Event Channels that the server can contain. It will only be limited by the resources of the Operating System that is being used. The criteria required to determine the number of Event Channels required will be determined by the software developers/system integrators application architecture. Some applications may have multiple event generating suppliers transferring events through a single channel. While others may have an Event Channel for every supplier.

The diagram below shows an example architecture in which suppliers and consumers communicate through three Event Channels contained within a single server.

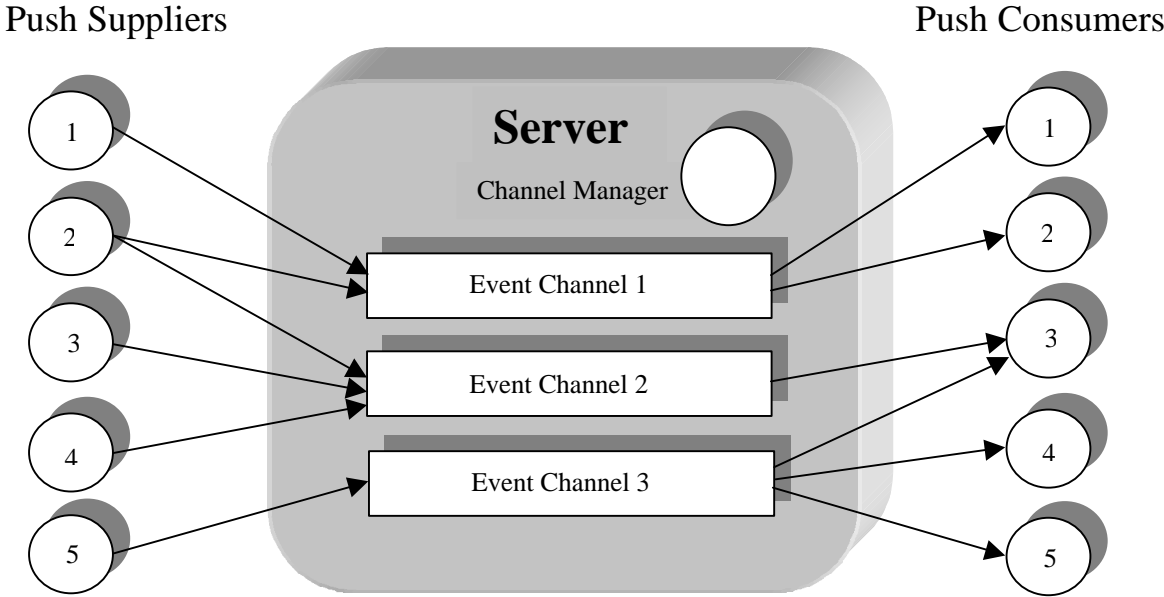


Figure 5-1 Suppliers and consumers using Event Channels

As shown in Figure 5-1, suppliers and consumers can be connected to multiple Event Channels. Push Supplier 2 is connected to Event Channel 1 and Event Channel 2, and Push Consumer 3 is connected to Event Channel 2 and Event Channel 3. It is also possible for suppliers and consumers to be connected to Event Channels in multiple servers as shown in the diagram below.

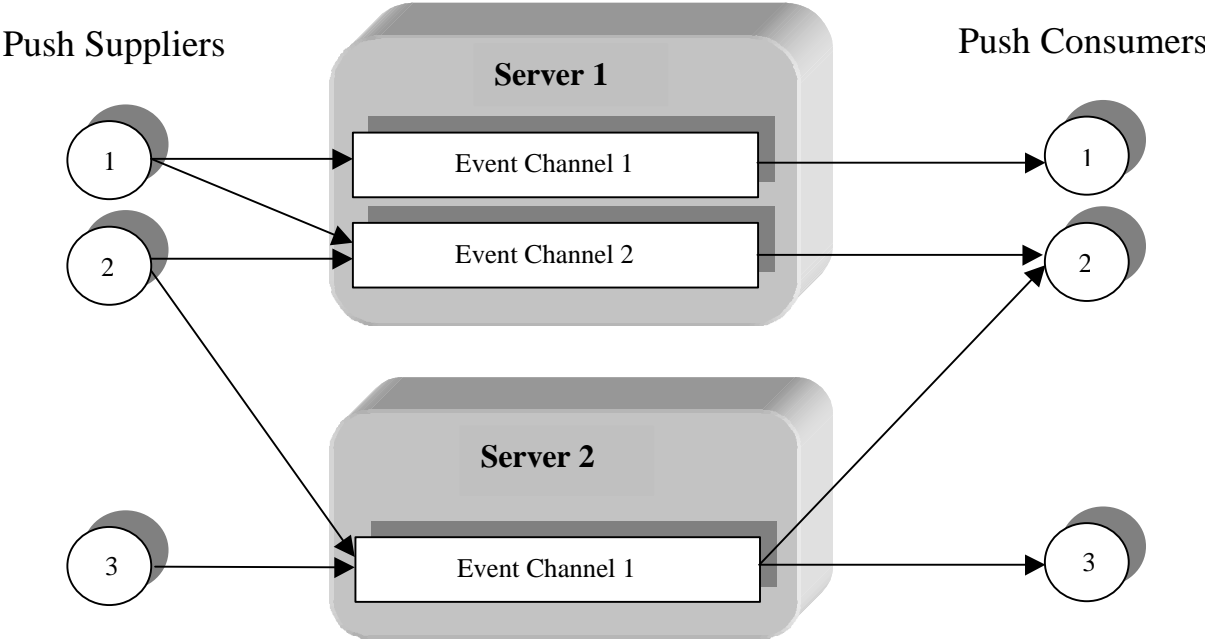


Figure 5-2 Suppliers and consumers using two different Servers

The server maintains up to 'n' Event Channel objects and a single ChannelManager object, that allows the software developer/system integrator to maintain Event Channels within the server.

The server application must be available at each host where the Event Channel is required.

Registering the Event Service in the Implementation Repository

Before being able to use the Event Server the software developer/system integrator must register the server application with the Implementation. The Event Server will have a number of command-line arguments, at the basic level it requires the mandatory channel identifier. The arguments associated with the Event Server are detailed below :

EventServer <Channel_id> [-rebind] [-server_name *name*] [-buffer *size*] [-timeout *seconds*]

EventSever	This is the executable name of the Event Server application.
Channel_id	Each channel in the Event Server must have a unique identifier, since the Event Server can contain multiple channels.
-rebind	When a server attempts to bind a name to an Event Channel object using the Naming Service, the Naming Service may indicate that the name is already bound. Using this switch the server will call the Naming Service rebind() operation to override the original binding.
-server_name	With this switch it is possible to specify a different server name, by default the Event Server name will be ES . The server must be registered in the Implementation Repository with the same name specified.

-buffer	Every Event Channel object will have an associated buffer, that determines the number of events a channel can store. The default will be 1000. Using this switch changes all the buffer sizes for all the Event Channels associated with a server application. To change the buffers of individual Event Channels the user will use the setChannelCapacity() method associated with every Event Channel object.
-timeout	By default, the server has an infinite timeout. Using this switch the server will timeout after the specified time in seconds.

Table 5-1 The command-line switches for the Event Server

Publishing the Channel Identifiers with the Naming Service

The server application will publish each Event Channel identifier with its associated Interoperable Object Reference (IOR) in the CORBA Naming Service. The Naming Service provides the CORBA compliant way to obtain IOR's. When the Event Server publishes the Event Channel identifier it will be located at one level below the Root Context (level 1) of the Naming Service. The Identifier and Kind attributes of the Naming Context has the same value as the Channel identifier. For example, suppose the user registers an Event Server that has three Event Channels; *channel_1*, *channel_2* and *channel_3* with the Implementation Repository. When the server executes, it will register these names with the Naming Service. The server will contain the following, see Figure 5-3, assuming that it has no other names registered. The Object (IOR) is placed directly below the naming context with the object name *eventChannel-eventChannel*.

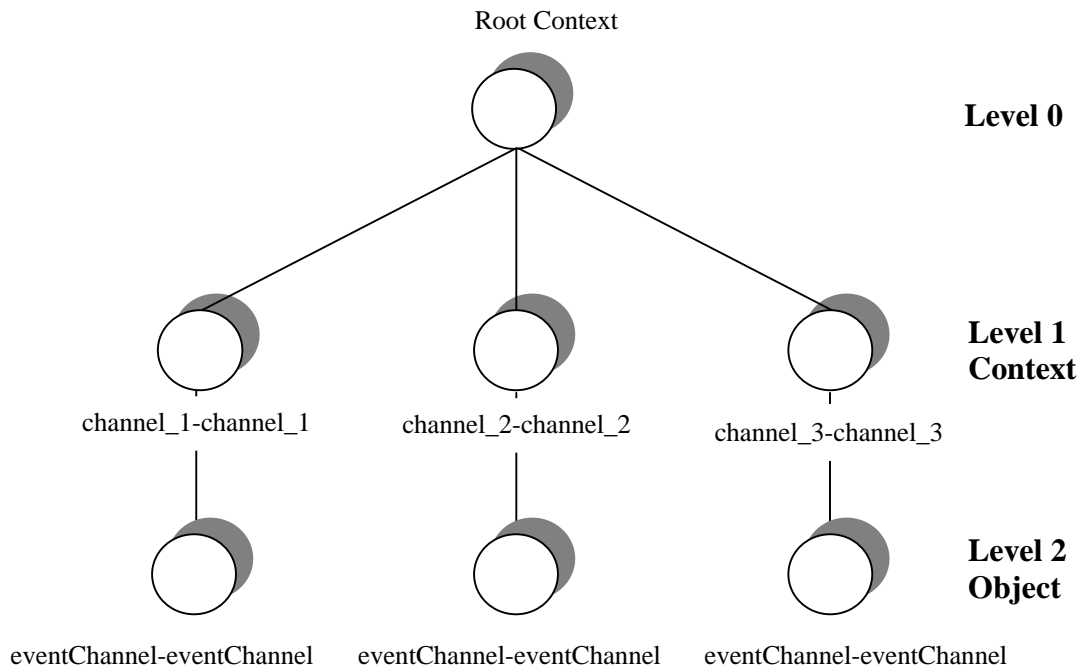


Figure 5-3 The Naming Service Context Tree layout

Figure 5-3 above, shows how the server will register the channels with the Naming Server. For all the channels, both the ID and Kind fields are shown for the naming context. The ID and Kind values are separated by a '-', for example; *channel_1-channel_1*.

It is possible that the Name Server could already have *channel_1-channel_1* registered at level 1 by another Event Server. If this is the case then the Event Server will throw an **AlreadyBound** exception and display the message to the user when the Event Server starts executing. This means that the Event Channel will not be registered with the Naming Service. If the Event Server only contains a single Event Channel (in this case *channel_1-channel_1*) then after the Event Server has displayed the message to the user it will exit. If however, the server was registered with *channel_1*, *channel_2*, and *channel_3* the Event Server will display the error message to the user about *channel_1*, but the application will not exit. Instead, the Event Server will contain two Event Channels, *channel_2*, and *channel_3* instead of three. It will be left to the user to correct the problem if they wish, by making the server application exit for this problem would only lead to unnecessary confusion.

However, it is possible to override the name that is already registered with the Name Server. Again, supposing *channel_1-channel_1* is already bound at level 1 within the Name Server. By using the **-rebind** switch at the command-line when registering the Event Server with the Implementation Repository means that the Event Server will force the Naming Service to replace the current binding for the name with the new binding. This will mean that the Event Server will not display any messages to the user, it is expected that the user will know the names registered with the Naming Service when using this switch.

Also, when the TCDEvents server is executed, it not only registers the channel names with the naming service but also the Channel Manager. This is so that the application developer or system integrator can obtain a handle to the Channel Manager object that will allow them to create and manipulate the Event Channels within the server. The Naming Service will contain the following, see Figure 5-4, assuming the Event Server is registered with a single Event Channel, *channel_1*.

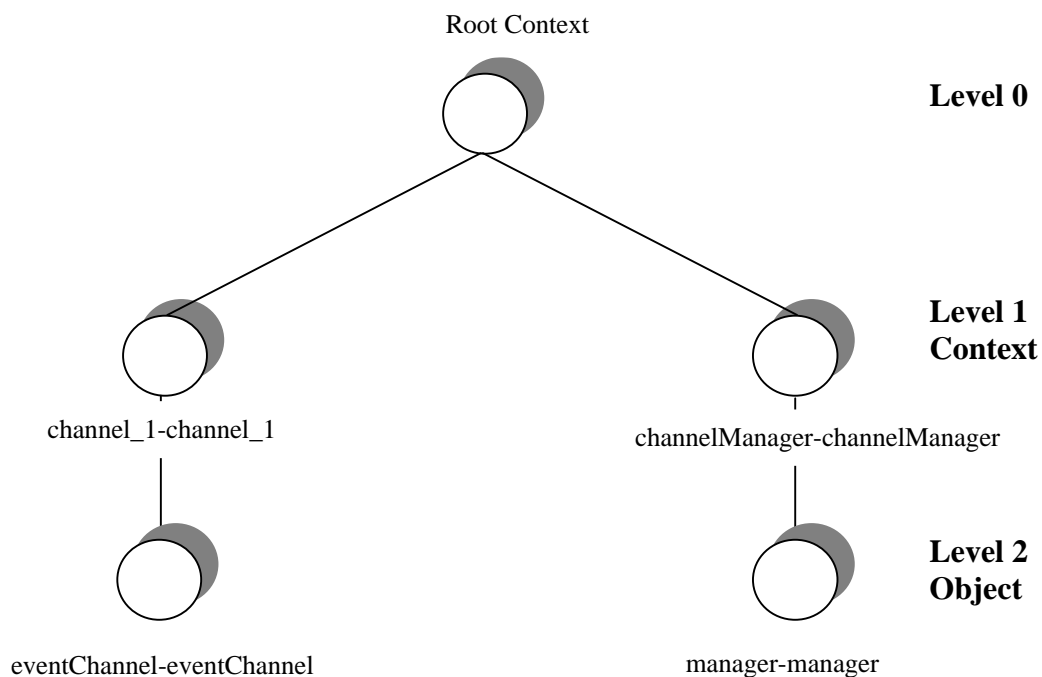


Figure 5-4 Channel Identifier and Manager in the Naming Tree

Event Channel

The Event Channel object (**EventChannelImpl**) is created automatically when the Event Server is started, for every Event Channel associated with the Event Server there is a corresponding **EventChannelImpl** object. The Event Channel acts as both, a consumer and a supplier and is the object that provides the decoupling between suppliers and consumers.

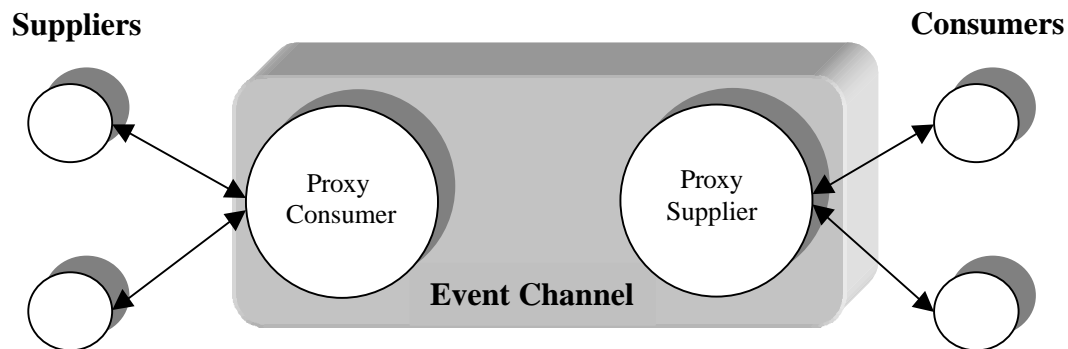


Figure 5-5 The Event Channel

The Event Channel keeps track of the Suppliers and Consumers connected to it by using Vectors. There are four Vector objects associated with the **Channel** object :

- push_consumers
- pull_consumers
- push_suppliers
- pull_suppliers

The **EventChannelImpl** object, created by the method *addChannelName()* of the **Channel** object, contains references to the **SupplierAdminImpl** and **ConsumerAdminImpl** objects. The **SupplierAdminImpl** object contains two methods used to create and manage Proxy Consumers; *obtain_push_consumer()* and *obtain_pull_consumer()*. One of these methods is invoked by the supplier application depending on the type of application; Push or Pull. Invoking the *obtain_push_consumer()* method creates a **ProxyPushConsumer** object which is then added to the push_consumers vector list. Invoking the *obtain_pull_consumer()* method creates a **ProxyPullConsumer** object which is then added to the pull_consumers vector list. Similarly, the **ConsumerAdminImpl** object contains two

methods, *obtain_push_supplier()* that creates a **ProxyPushSupplier** object and adds it to the *push_suppliers* vector list, and *obtain_pull_supplier()* that creates a **ProxyPullSupplier** object and adds it to the *pull_suppliers* vector list.

When a supplier or consumer application no longer wishes to be connected to the Event Channel it must call the appropriate disconnect method to remove its reference from the correct vector list. There are four different disconnect methods, depending on the application type :

1. *disconnect_push_consumer()*

This is a member of the **ProxyPushConsumerImpl** object, when called it removes its reference from the *push_consumers* vector list.

2. *disconnect_pull_consumer()*

This is a member of the **ProxyPullConsumerImpl** object, when called it removes its reference from the *pull_consumers* vector list.

3. *disconnect_push_supplier()*

This is a member of the **ProxyPushSupplierImpl** object, when called it removes its reference from the *push_suppliers* vector list.

4. *disconnect_pull_supplier()*

This is a member of the **ProxyPullSupplierImpl** object, when called it removes its reference from the *pull_suppliers* vector list.

For every Event Channel there is an **EventChannelImpl** object which is created by the *addChannelName()* method of the **Channel** object. Invoking this method, the reference of **EventChannelImpl** is added to an *eventChannelArray* vector. This vector is used to ensure that the TCDEvents server keeps track of the Event Channels it has. When a channel is created by the **ChannelManager** object its reference is also added to the vector. Calling the *destroy()* method of **EventChannelImpl** calls the *removeEventChannel()* method of the **Channel** object and removes the corresponding reference from the *eventChannelArray* vector.

Associated with each Event Channel is a synchronized wait queue. The queue is used to store the event data generated by suppliers before being delivered to consumers.

The use of the queue means that event data does not have to be delivered to

consumers as soon as a supplier has generated an event. It also minimizes resource utilization by restricting the number of events in transit. To ensure thread safety, the queue is fully synchronized and implements a wait notify mechanism. This ensures that event data does not get lost when a supplier tries to add an event item to a full queue, and the Event Channel does not try to remove an event item from an empty queue.

The **EventChannelImpl** object has two methods to enable the user to manipulate the size of the queue. The synopsis for these methods are :

```
public void setChannelCapacity( int capacity);  
public final int getChannelCapacity();
```

The method *setChannelCapacity()* allows the user to control the size of the internal queue that is associated with every Event Channel, by default the size is 1000. The parameter for this function is the value that the buffer size will be set to. The method *getChannelCapacity()* returns the size of the queue.

The delivery order to consumers is beyond the scope of this project. Applications that require a specific order will have to use the forthcoming CORBA Message Service. The Quality of Service supported for this Event Service is “best-effort”, application domains that require highly levels of reliability will have to use the CORBA Notification Service.

Channel Manager

The Event Service implementation provides an Event Channel administration interface, *ChannelManager*, defined in the IDL module *EventServerAdmin*. Every server application will have a ChannelManager object. The purpose is to allow the software developer/system integrator to create and manipulate multiple Event Channels within the server application.

The ChannelManager object in the Event Server is named “*ChannelManager*”. The user is able to obtain its reference by invoking the method *resolveChannelMgr()*. The server application will automatically register the ChannelManager with the Naming Service when the server is started. Appendix A shows the IDL module for the ChannelManager.

A description of the functions associated with the ChannelManager object are described below :

Functions of the ChannelManager object	
Function	Description
createChannel(String channel_name)	Creates an Event Channel within the server application and registers the name identified by <i>channel_name</i> , with the Naming Service.
findChannel(String channel_name)	Finds the Event Channel associated with the channel name <i>channel_name</i> .
destroyChannel(String channel_name)	Destroys the Event Channel and removes the Channel’s object reference from the Naming Service.
listAllChannels()	Returns a list of all the Event Channels associated with the server application.
findChannelName(Channel channel_ref)	Finds the Channel that has the object reference, <i>channel_ref</i> , and returns the name of the channel to the calling function.

Table 5-2 The ChannelManager methods

The Push Model for Untyped Events

There are four IDL interfaces that are used to support connection to and disconnection from the Event Channel for the Push model :

- *PushSupplier*
- *PushConsumer*
- *ProxyPushConsumer*
- *ProxyPushSupplier*

The interfaces *PushSupplier* and *ProxyPushConsumer* allow suppliers to supply event data to the Event Channel. The *PushConsumer* and *ProxyPushSupplier* interfaces are specific to consumers and allow them to receive event data.

Connecting a Supplier

The supplier application obtains the object reference for the required Event Channel it wants to be connected to by using the Naming Service. The supplier invokes the *for_suppliers()* method of the **EventChannelImpl** object. This returns a reference to the **SupplierAdmin** object, that allows the supplier to obtain an Event Channel consumer proxy. The supplier invokes the *obtain_push_consumer()* method of the **SupplierAdminImpl**, which creates a **ProxyPushConsumer** object, adds the reference to the *push_consumers* vector and returns the reference. The supplier application finally connects to the Event Channel by invoking the *connect_push_supplier()* method of the **ProxyPushConsumerImpl** object passing it the object reference to the Event Channel. This allows the Event Channel to be disconnect from the supplier.

Connecting a Consumer

The consumer application obtains the object reference for the required Event Channel it wants to be connected to by using the Naming Service. The consumer invokes the *for_consumers()* method of the **EventChannelImpl** Object. This returns a reference to the **ConsumerAdmin** object, that allows the consumer to obtain an Event Channel supplier proxy. The consumer invokes the *obtain_push_supplier()* method of the

ConsumerAdminImpl object, which creates a **ProxyPushSupplier** object, adds its reference to the `push_suppliers` vector list. It finally creates and starts the **ThreadPool** that is associated with a thread group before returning the reference to the *ProxyPushSupplier* object. When starting the **ThreadPool** it determines the number of consumer threads. For each thread the **ConsumerThread** object starts the consumer thread running. Each **ConsumerThread** object is also associated with the thread group. The purpose of the **ConsumerThread** object is to take an event from the queue and deliver it to all the connected consumers. If there are no suppliers connected to the Event Channel, or if there are no events in the queue, the thread sleeps for two seconds. Once the thread has completed its task, it sleeps for a few milliseconds, this allows any other waiting threads to run. The consumer application finally connects to the Event Channel by invoking the `connect_push_consumer()` method of the **ProxyPushSupplier** object passing its object reference to the Event Channel so the Event Channel can use it to push events to the consumer.

Transfer of Untyped Events

Once the setup of both, the supplier and the consumer is complete, the supplier can start transmitting events to the consumer via the Event Channel. To transmit an event, the supplier application invokes the `push()` method from the **ProxyPushConsumerImpl** object which adds the event to the queue. One of the waiting threads in the Consumer Thread Pool will take the event item from the queue and deliver the event to all the consumers connected to the Event Channel. As soon as this happens, the thread will again enter the waiting state for the next event. It is highly possible that more than one thread will want to take the same item from the queue at the same time. To overcome the conflict to a shared data item at the same instance, the “*take item*” method for the queue is synchronized. This means that only one thread at a time can access this method, all other threads that want to take items from the queue will have to wait until the thread has relinquished control of the “*take item*” method.

The consumer thread pool is controlled by Java’s Thread Group class which is used for managing and manipulating groups of threads. Using threads means the Event Channel benefits from the advantages of concurrent programming, increased speed

and minimized resource utilization. This is very important when using events in a distributed system, as the events have to be handled and delivered in the most cost effective way. Figure 5-6 below, shows the conceptual view of the thread pool and the queue that is associated with each Event Channel.

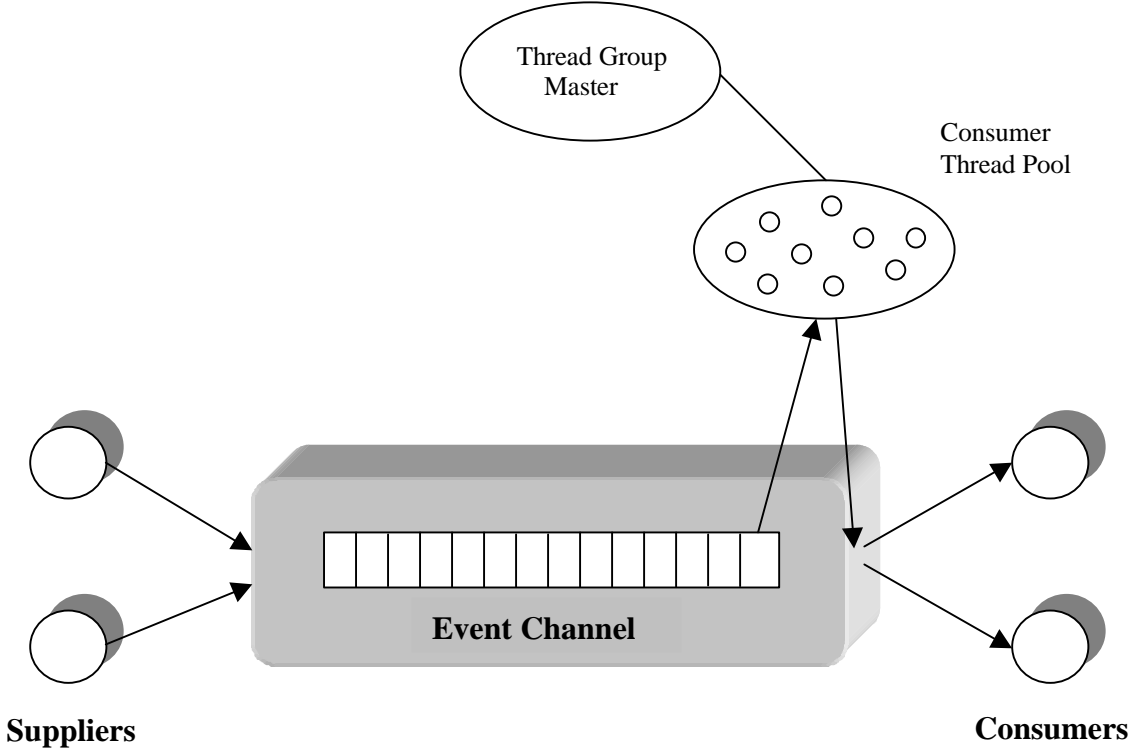


Figure 5-6 The Push Model Architecture

The Pull Model for Untyped Events

There is a similar set of IDL interfaces that support connection to and disconnection from the Event Channel for the Pull model:

- *PullSupplier*
- *PullConsumer*
- *ProxyPullConsumer*
- *ProxyPullSupplier*

The *PullConsumer* and *ProxyPullSupplier* are the interfaces that allow consumers to request event data from the Event Channel. The *PullSupplier* and *ProxyPullConsumer* interfaces allow the Event Channel to request event data from suppliers.

Connecting a Supplier

The first thing the supplier application does, is to obtain an object reference of the required Event Channel, by using the Naming Service. The supplier invokes the *for_suppliers()* method of the **EventChannelImpl** object. This returns a reference to the **SupplierAdmin** object, that allows the supplier to obtain an Event Channel consumer proxy. The supplier invokes the *obtain_pull_consumer()* method of the **SupplierAdminImpl** object, which creates a **ProxyPullConsumer** object. It adds the reference to the *pull_consumers* vector and returns the reference. For every **ProxyPullConsumerImpl** object that is created (one per supplier), it spawns another process. The purpose of this process is to contact each supplier connected to the Event Channel and to ask if it has an event. If a supplier has an event, it is then added to the queue. The supplier application finally connects to the Event Channel by invoking the *connect_pull_supplier()* method of the **ProxyPullConsumerImpl** object.

Connecting a Consumer

The consumer obtains a reference to the required Event Channel in the normal way. The consumer invokes the *for_consumers()* method of the **EventChannelImpl** object. This returns a reference to the **ConsumerAdmin** object, that allows the consumer to

obtain an Event Channel supplier proxy. The consumer invokes the *obtain_pull_supplier()* method of the **ConsumerAdminImpl** object which creates a **ProxyPullSupplier** object and adds its reference to the *pull_suppliers* vector before returning the reference. The consumer application finally connects to the Event Channel by invoking the *connect_pull_consumer()* method of the **ProxyPullSupplierImpl** object. It passes its object reference to the Event Channel so the Event Channel can use it to call the methods *pull()* or *try_pull()* to obtain events.

Transfer of Untyped Events

Once a supplier and a consumer have been connected to the Event Channel the transfer of events can take place. In the Pull model the consumer initiates event transfer. The consumer can either invoke the *pull()* or *try_pull()* methods of the *PullSupplier* object.

- *pull()*

The consumer invokes this method on the **ProxyPullSupplierImpl** object. If the Event Channel does not have any event data, then the **ProxyPullConsumerImpl** object has a spawned process that will invoke a *pull()* method operation on every supplier that is connected to the Event Channel. If the supplier has generated an event, then it is added to the queue and pulled by the consumer application. The *pull()* method will block until a supplier has generated an event, causing the consumer application to block until it has received the event.
- *try_pull(Boolean has_event)*

The consumer invokes this method on the **ProxyPullSupplierImpl** object. If the Event Channel does not have any event data, then, as for the *pull()* method, the **ProxyPullConsumerImpl** object contacts all the connected suppliers asking for an event, which is then added to the queue. This method, unlike the *pull()* method, does not block. If a supplier has event data it sets the *has_event* parameter to *true* and returns the event data to the consumer. If the supplier does not have any data then the parameter *has_event* is set to *false*.

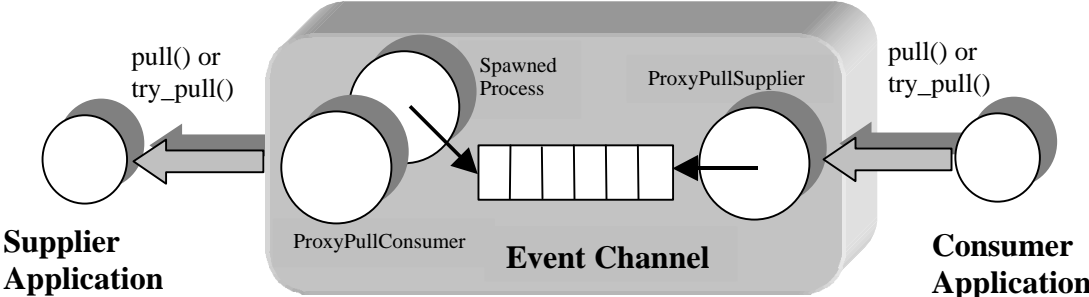


Figure 5-7 The Pull Model Architecture

Locating the Event Channel and Channel Manager

To obtain references to either the Event Channel or the Channel Manager objects that have been registered with the Naming Service, TCDEvents supplies a class called **ResolveNames**. The software developer or system integrator will create an instance of **ResolveNames** in both, the supplier and the consumer application. The **ResolveNames** object contains two methods :

- ***public EventChannel resolveChannelName(String channelName)***
This method takes the Event Channel name that needs to be resolved. It is the name of the channel identifier that the user specified when either registering the server application with the Implementation Repository or creating the Event Channel with the Channel Manager object. It returns the Event Channel's IOR, a reference to the **EventChannel** object. This method will be invoked in both, the supplier and the consumer application before suppliers or consumers can be connected to the channel.
- ***public ChannelManager resolveChannelMgr()***
Since there is only one Channel Manager object per server, this method does not take the name as a parameter as it will always be "ChannelManager". It returns a reference to the Channel Manager's IOR, the **ChannelManager** object. The place to invoke this method depends on the application domain of the user, it is to be called by the consumer or the supplier. It is very unlikely that this function will be called by both, a supplier and a consumer application. Normally, only one of these applications will be used to control and manipulate the Event Channels of the server.

Resolving the Event Channel's and the Channel Manager's names to obtain their object references is very similar. Below is the general outline of how this is achieved for both of these functions.

1. Obtain a reference to the Naming Service by calling the CORBA function *resolve_initial_references()*.

2. Create a CosNaming name, to build up the Identifier and Kind fields for the Name. For example, suppose the name of the Event Channel is *channel_1* then the Identifier field of the naming context would be *channel_1* and the Kind field would also be *channel_1*. The Identifier and Kind values are always the same for the Event Service. The same principle applies to the Channel Manager's name.

3. From the CosNaming name, you then resolve the name from the root context, if the Naming Service contains such a name, it returns the object reference of that name. Otherwise, a null object reference is returned.

The Consumer/Supplier Wizard

One of the problems of the Event Service is the multiple steps required to connect a supplier and a consumer application to an Event Channel. These steps are necessary for every application in a distributed system that wishes to use the Event Service. It would be a good idea to remove this type of burden from the software developer/system integrator by having a Java GUI tool that generates code for doing this.

The GUI tool is a wizard application, that steps the user through each process in a simple and easy to use manner. The user is not able to progress to the next screen until they have completed all the necessary information that is required. Once the process is totally complete, the wizard tool generates the Java code to connect the supplier/consumer to an Event Channel. All the user has to do is to send the event data from the supplier and receive the event data at the consumer. Once the generated files are compiled the user can use the Event Service and take advantage of its benefits. Below are the screens that form the wizard application.

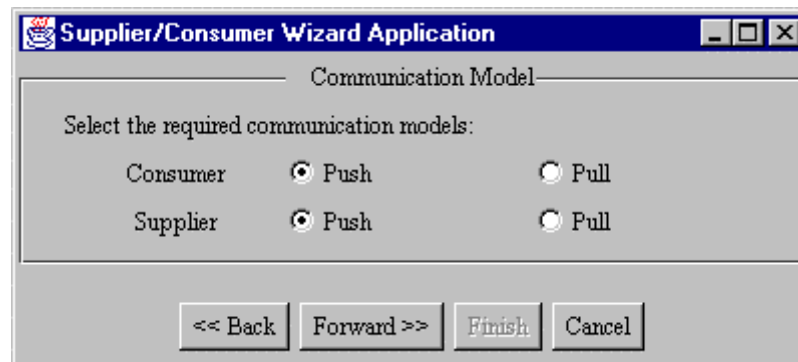
1. Selecting the Application Type



This is the first screen for the wizard application. By default, both the selection boxes of the consumer and the supplier are selected. The user is able to deselect/select the boxes by using the mouse button. The Back and Finish buttons are disabled, the Forward button is only disabled if neither option is selected.

This will prevent the user from moving to the next stage. The Cancel button, if selected will undo any changes and close the wizard application.

2. Selecting the Communication Model



The user selects the type of communication model for the supplier/consumer. By default the Push buttons are selected for both, supplier and consumer. The Finish button is disabled.

3. Naming the Consumer/Supplier Files



This screen allows the user to enter the names of the consumer and supplier files that will get generated. The Back button is enabled, allowing users to return to the previous screen if they wish. The Forward button only becomes enabled when a file name has been entered for the supplier and consumer. The Finish button is disabled.

4. Event Channel Information



Supplier/Consumer Wizard Application

Event Channel

Enter the Event Channel information:

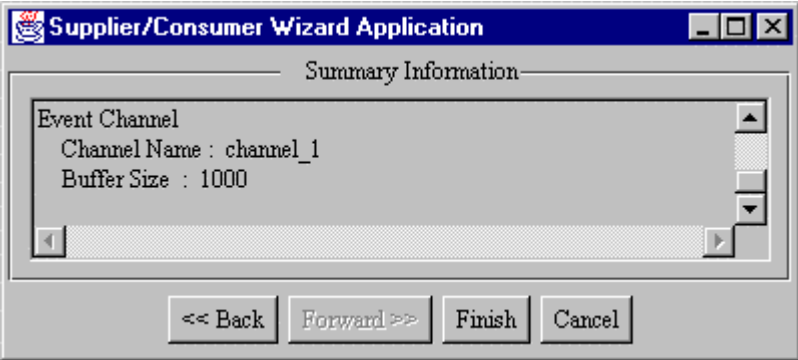
Channel Name: channel_1

Buffer Size: 1000

<< Back Forward >> Finish Cancel

Screen four of the wizard allows the user to specify the Channel Name that they wish to connect to. Once a name has been entered, the Forward button becomes enabled. It also allows the user to specify a different buffer size, initially it displays the default value of 1000. If this field is deleted by the user the default size will be assumed. The Finish button is disabled.

5. Conformation Screen



Supplier/Consumer Wizard Application

Summary Information

Event Channel

Channel Name : channel_1

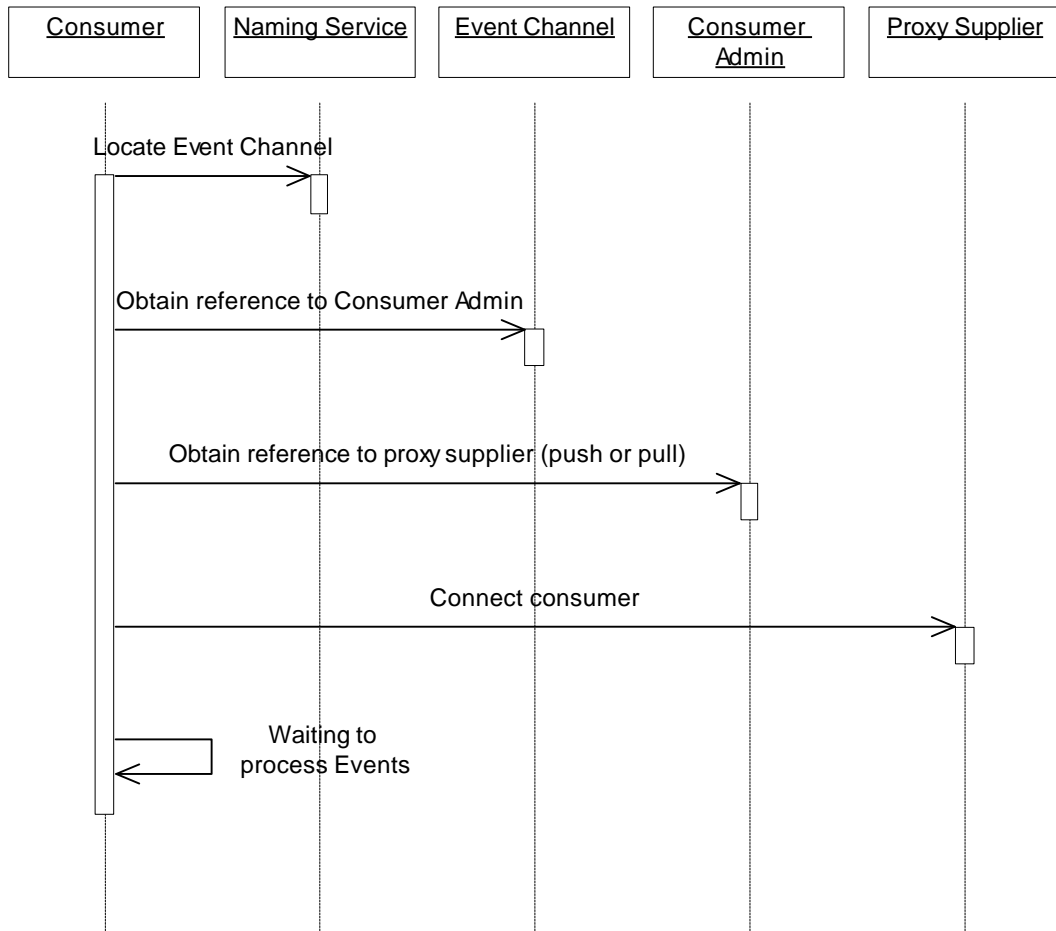
Buffer Size : 1000

<< Back Forward >> Finish Cancel

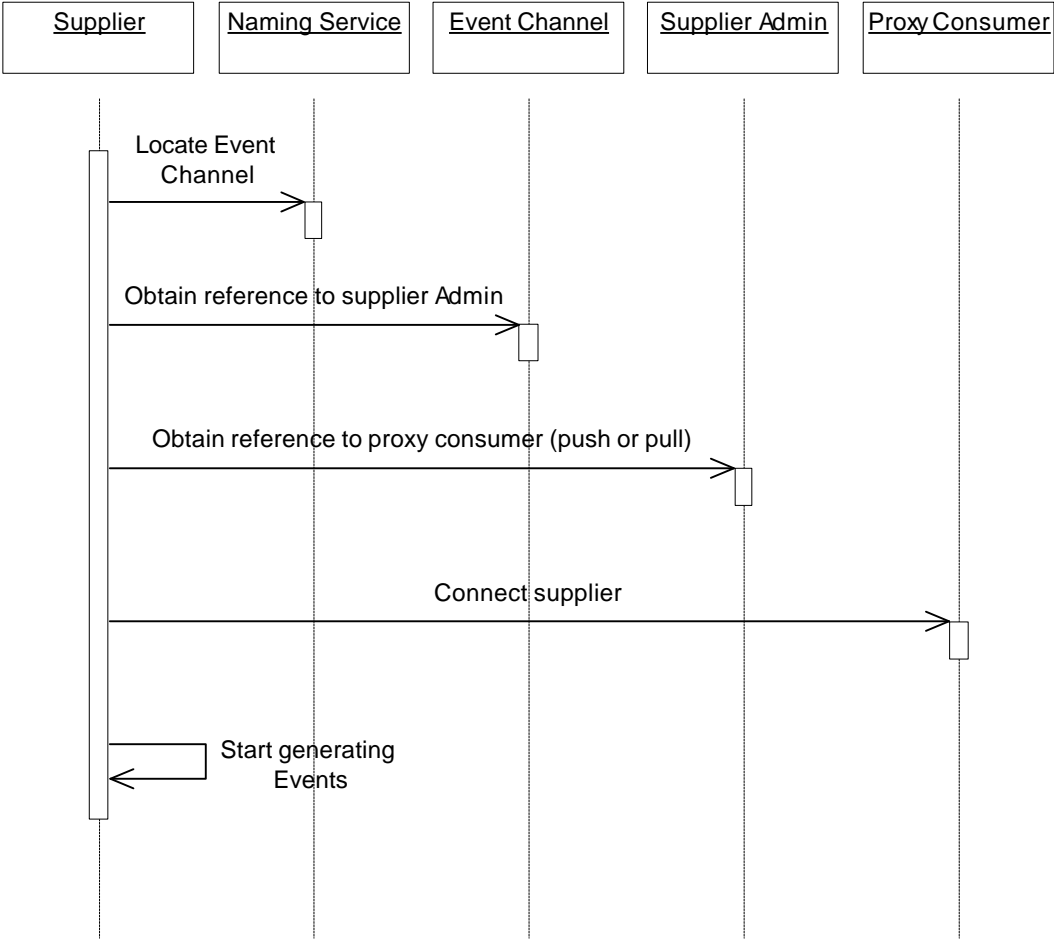
This is the final screen of the application wizard. It displays a summary of the information selected by the user about the supplier, consumer and Event Channel. The type of information displayed is, Event Channel name, buffer size, communication model for the supplier and the consumer and the file names for the supplier and the consumer. For this screen, the Forward button is disabled and the Finish button is enabled. Once the user is satisfied with their selection, the Finish button is selected and the code for the supplier and the consumer is automatically generated. The application then exits.

Sequence Diagrams

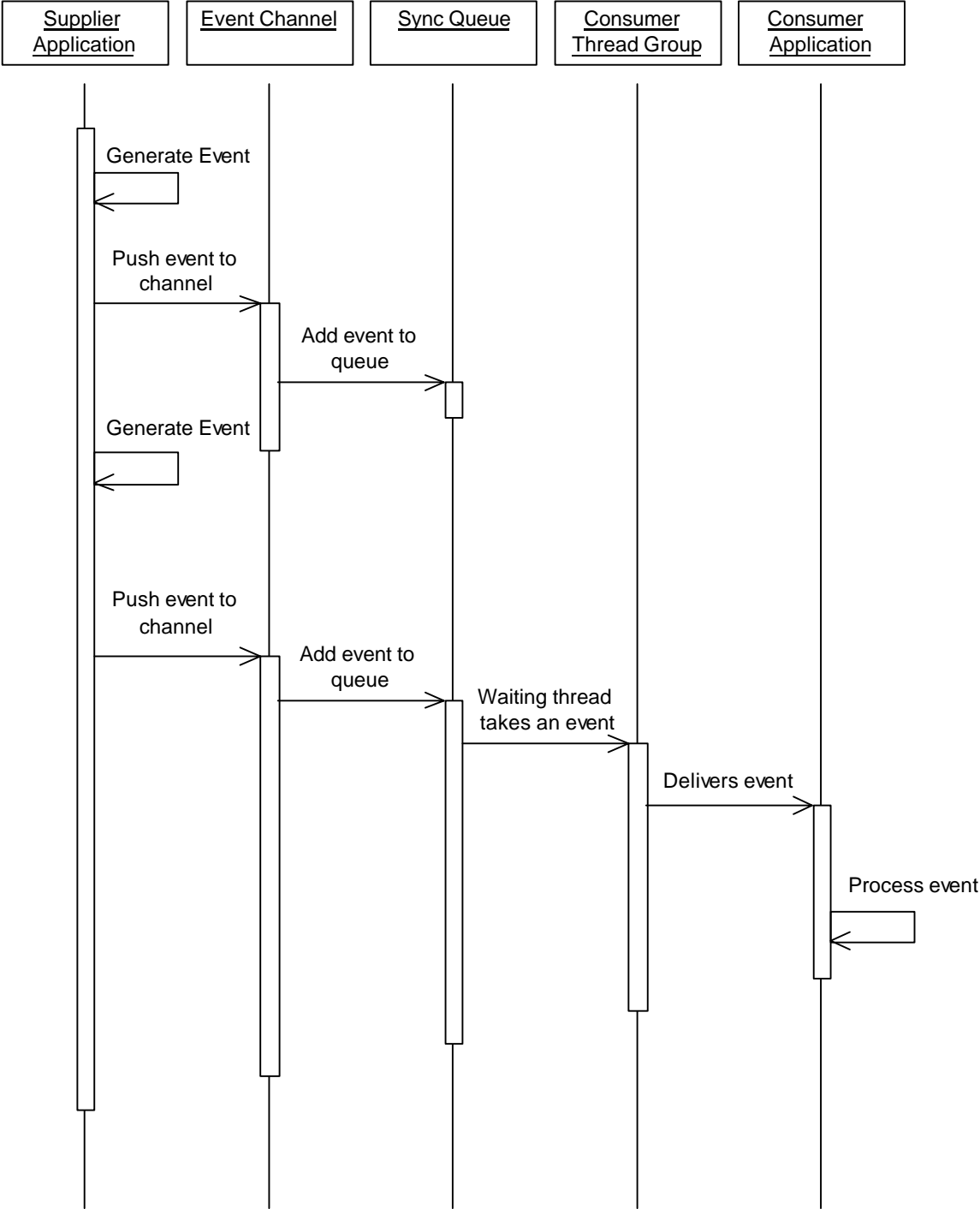
1. Connecting a Consumer to the Event Channel



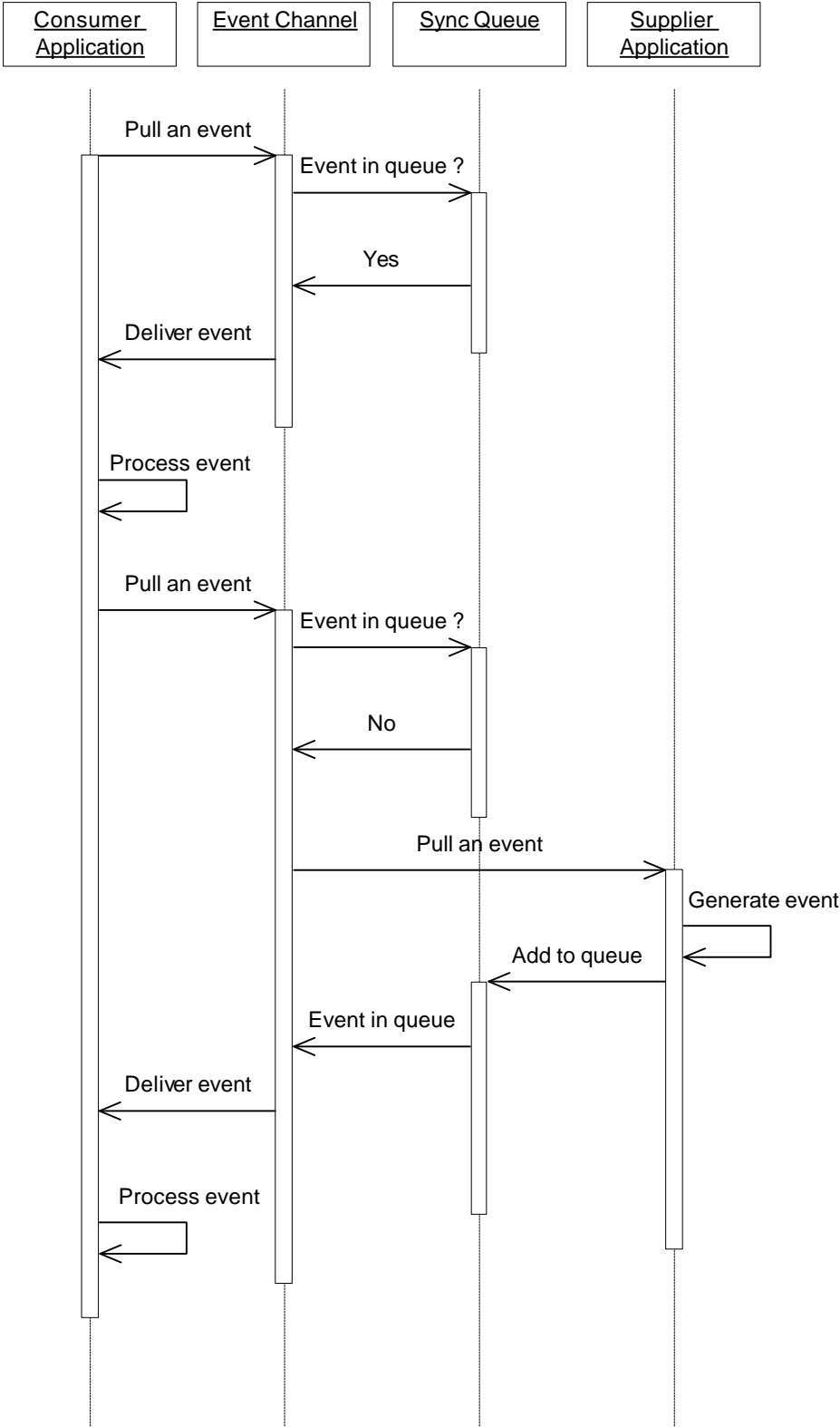
2. Connecting a Supplier to the Event Channel



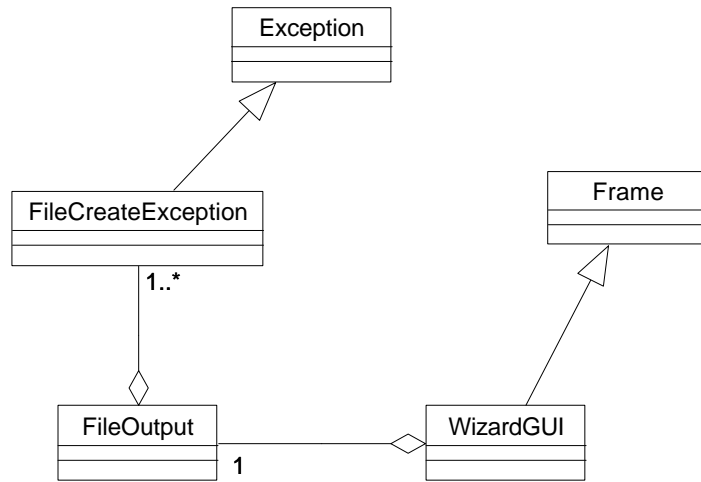
3. Supplying an Event to a Push Consumer



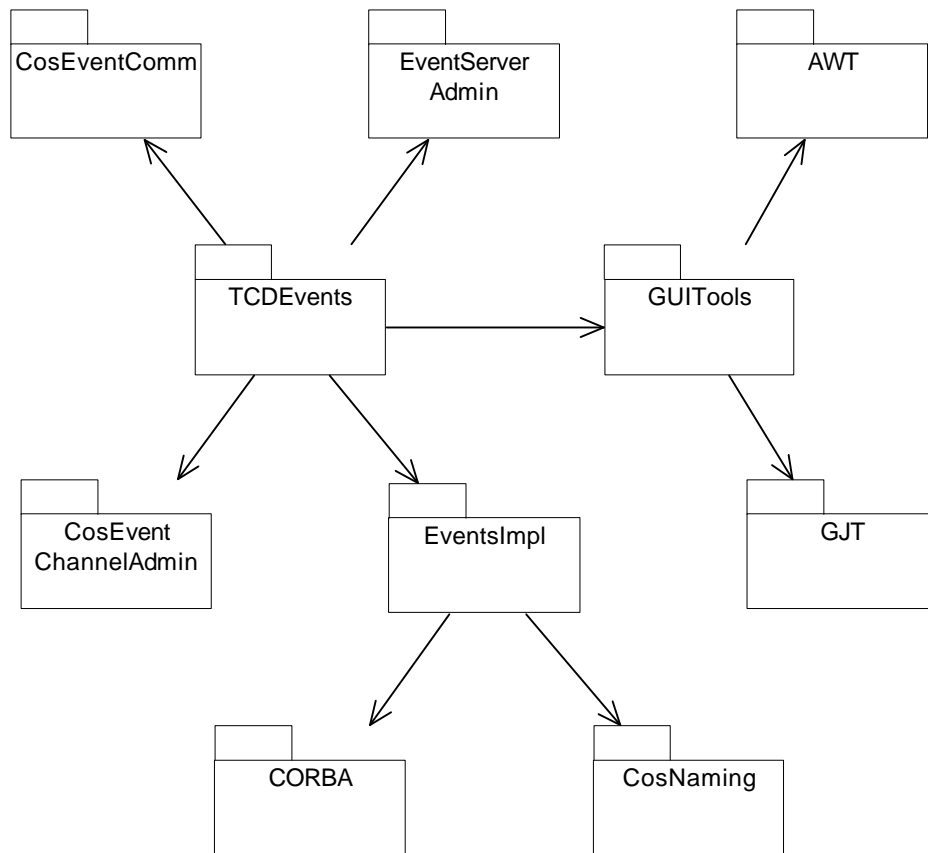
4. Supplying an Event to a Pull Consumer



- **GUI Wizard**



Package Diagram



Summary

TCDEvents has been designed so that it is easy to use with the minimal amount of effort and removes as much as possible the programming burden away from the software developer. It provides a server application which contains a Channel Manager object used for creating and manipulating the Event Channels. A GUI wizard application is used to automatically generate the supplier and the consumer applications.

6

Evaluation of TCDEvents

This chapter benchmarks TCDEvents with other CORBA and non-CORBA event models that have been implemented in Java. The main objective of the evaluation is to compare TCDEvents with other available event models in order to determine its strengths and weaknesses.

In the evaluation, TCDEvents was compared against two CORBA Event Services and one non-CORBA Event Service :

CORBA Event Services

1. jEVENTS [OUT:97] – This is a Java implementation for untyped events that supports both, push and pull style communication.
2. ORBacus [OOC:98] – This is an actual ORB, written in Java, that has an implementation of an untyped Event Service.

Non-CORBA Event Service

1. iBus [iBus:97] – An alternative event model to CORBA, that supports push and pull style communication.

The selection criteria for the event models to be used was chosen for a number of different reasons :

1. Availability – It had to be easily available either on an evaluation period or free to academic institutions.
2. Java – The models had to be written in Java. Although OrbixEvents, a C++ version, was considered, the availability of the product was very poor and there was not enough time to install the correct version.
3. Implementation – All models had to support both, push and pull style communication for untyped events.
4. Installation – All the models had to be easy to install with the minimal amount of effort.
5. OrbixWeb – The CORBA models had to be interoperable with IONA's OrbixWeb product.

Once the event models to be evaluated against TCDEvents were selected, the next stage was to ask a set of questions. These questions needed to be answered in order for the evaluation to be completed and conclusions to be drawn. The questions were :

1. How easy is it for the software developer/system integrator to use the Event Service. What type of coding is required.
2. Is it interoperable.
3. How do supplier and consumer applications contact the Event Channel.
4. Does the Event Service provide a server application.
5. Can the server contain multiple Event Channels.
6. How long does it take to connect a push consumer.
7. How long does it take to connect a push supplier.
8. How long does it take for a push supplier to generate 500 events.
9. How long does it take for a push consumer to consume 500 events.
10. How long does it take to connect a pull consumer.
11. How long does it take to connect a pull supplier.
12. How long does it take for a pull supplier to generate 100 events.
13. How long does it take for a pull consumer to consume 100 events.
14. What is the maximum performance of TCDEvents for the Push model.

All tests were carried out on a single Solaris machine running SunOS version 5.6. The CORBA event models used IONA's OrbixWeb Professional version 3.0. Each test was run 50 times, if the Event Service being tested required a server application running, then this was left running for the duration of the test.

The supplier and consumer applications were basically the same. A supplier would generate an event that was the current date and time, a consumer would receive the event and display it to the standard output of an Xterm console. The supplier and consumer applications for the CORBA models were identical, except for the methods used to send data to the Event Channel or receive data from the Event Channel. It was important to use the same code for supplier and consumer to ensure that no inconsistencies occurred between the different CORBA Event Services. The non-CORBA model (iBus) was a little different since it did not support the standard OMG Event Service API, the same code could not be used. Great care was taken to ensure that the generating and processing of the event information at both the supplier and consumer was as identical as possible with the CORBA models. This was to ensure there were no discrepancies between the tests.

Nine tests were carried out in total, the Push and Pull model's consisted of four tests each. There was also a single test carried out on the Push model of TCDEvents, to measure its performance.

The Push Model Test

Consisted of four tests, each test was run 50 times, while the server application was left running. Each test had an error of +/- 5% of the recorded values.

- **Connecting a Push consumer to an Event Channel**

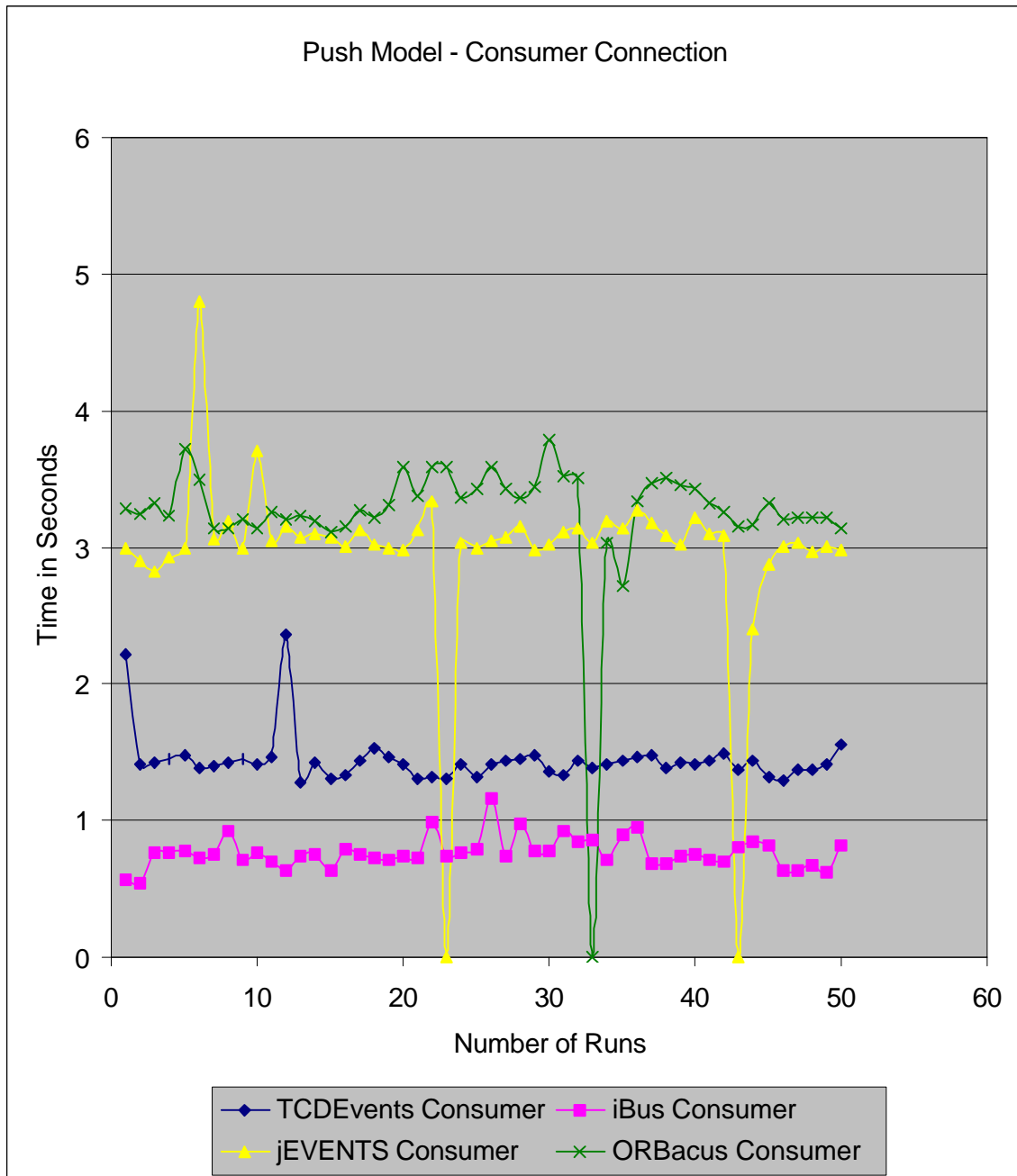


Figure 6-1 Connection of a Push Consumer

Figure 6-1 shows the plotted results from the test. TCDEvents was much quicker than the other CORBA models, jEVENTS and ORBacus, while iBus (non-CORBA model) was the fastest of all.

- **Connecting a Push Supplier to an Event Channel**

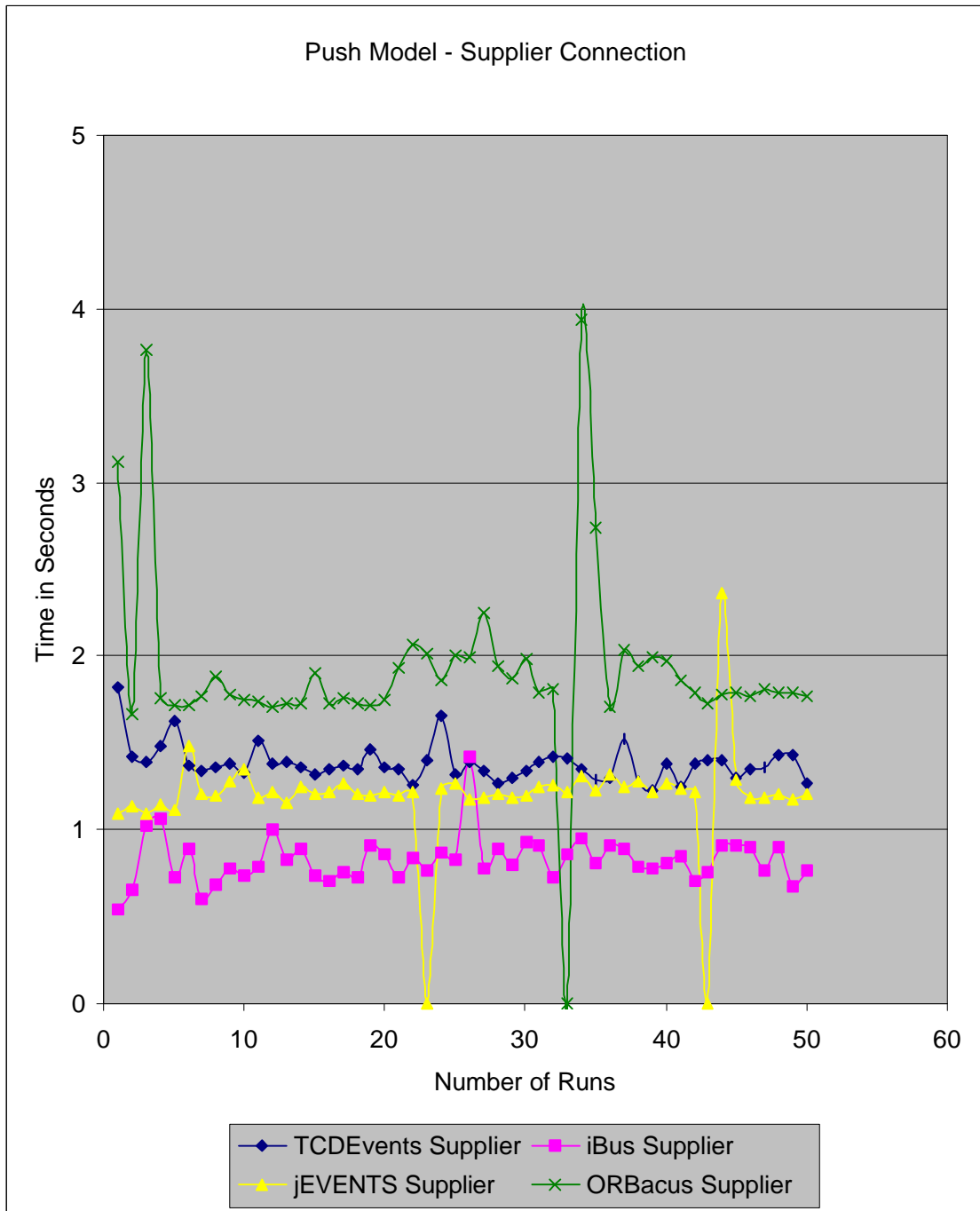


Figure 6-2 Connection of a Push Supplier

The connection times for TCDEvents and jEVENTS are almost identical, with ORBacus being the slowest of the CORBA models again. Once again iBus has the best connection times.

- **Push Supplier sending 500 Events**

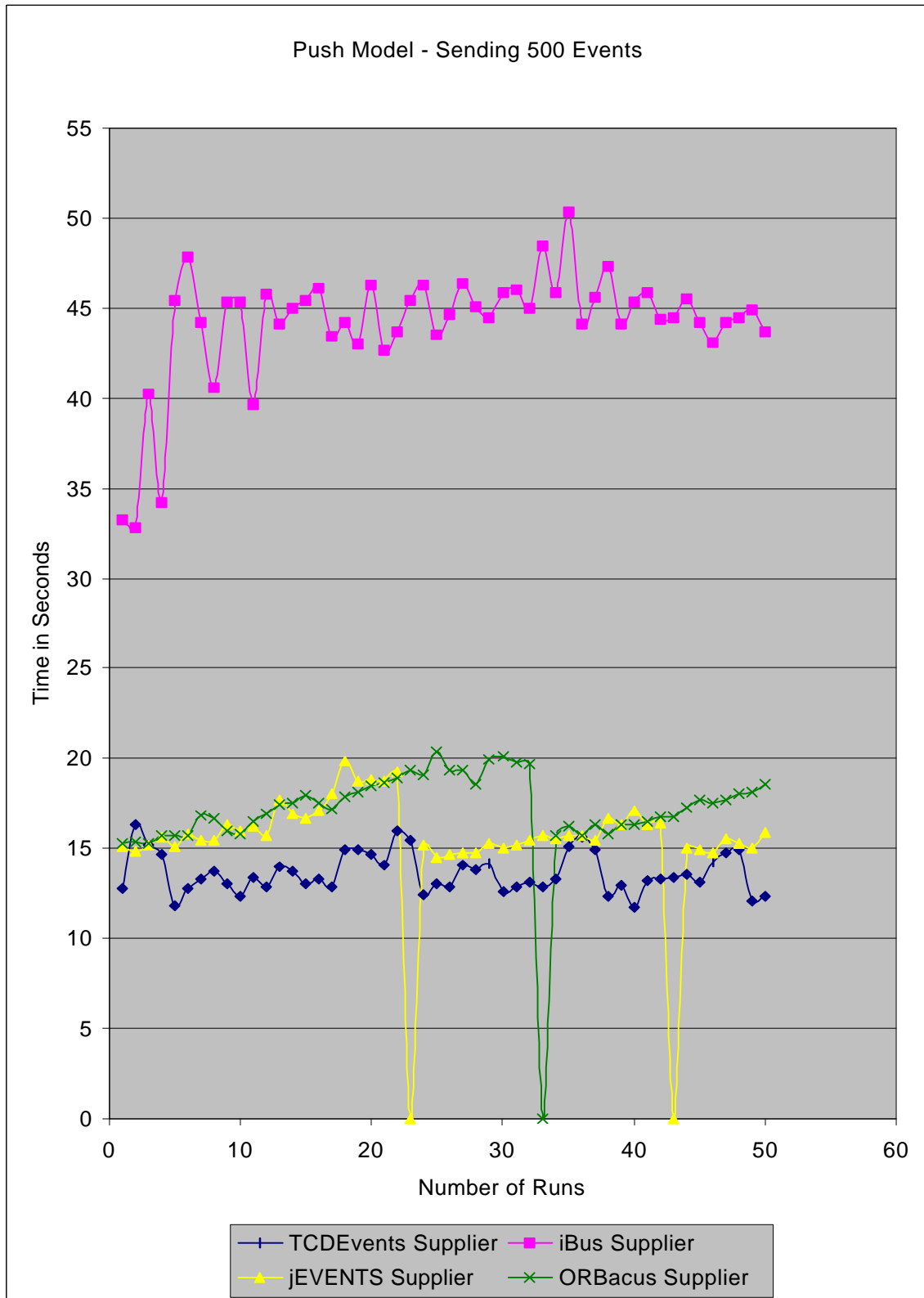


Figure 6-3 Push Supplier sending 500 Events

- **Push Consumer Receiving 500 Events**

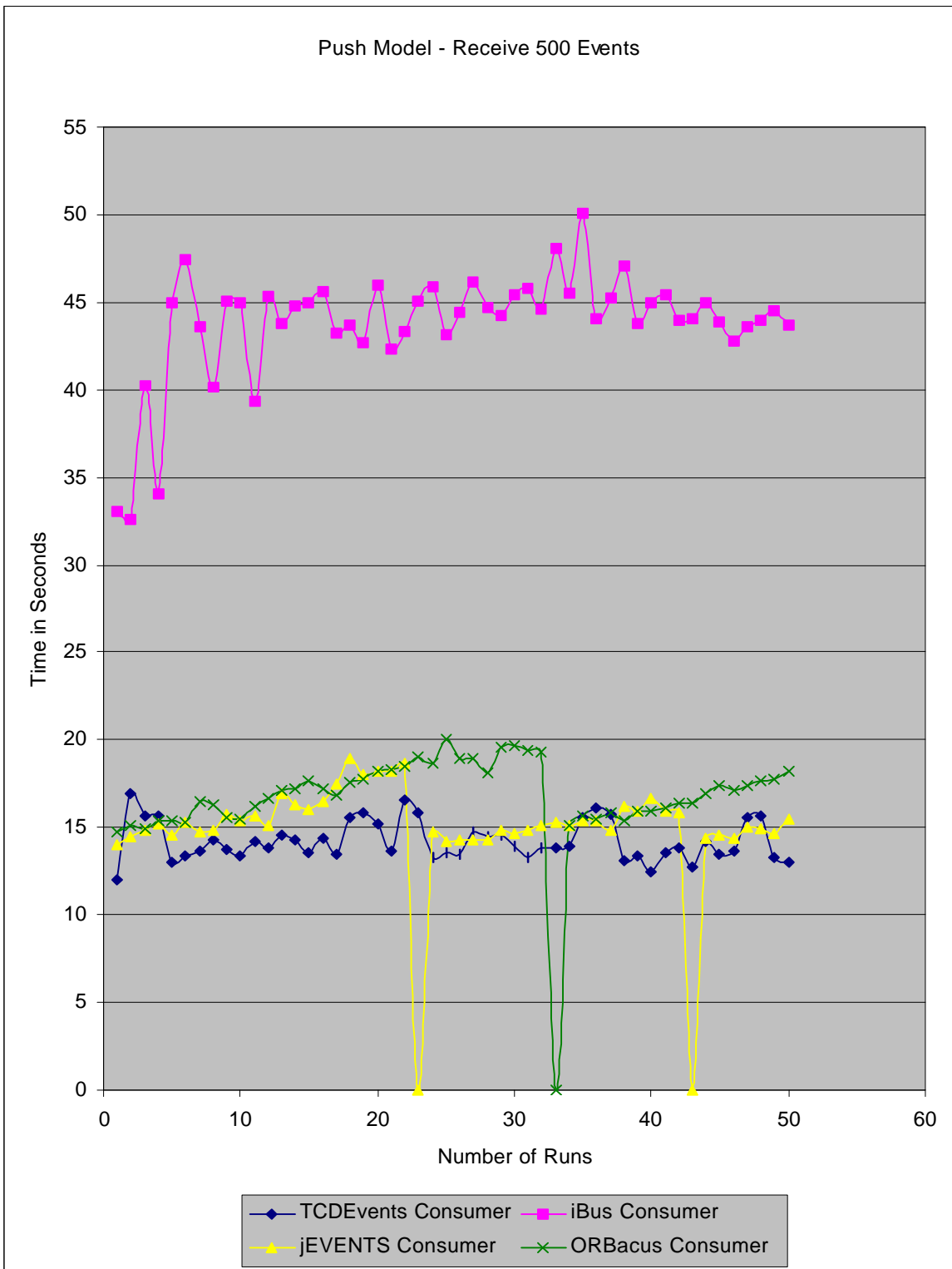


Figure 6-4 Push Consumer receiving 500 Events

In figure 6-3 the CORBA models take a similar amount of time, with TCDEvents being slightly faster. The non CORBA model, iBus is considerably slower.

Figure 6-4 shows results that are very similar.

Average Times for Push Model Tests

Average time in Seconds Over 50 Runs				
	Connect Consumer	Connect Supplier	Send 500 Events	Receive 500 Events
TCDEvents	1.44104	1.38278	13.64258	14.23216
jEVENTS	2.97114	1.19466	15.42218	14.9294
ORBacus	3.25254	1.92208	17.08714	16.72344
iBus	0.76418	0.82628	44.13258	43.82546

Table 6-1 Average test times for the Push model

Push Model Test Conclusions

The jEVENTS push model system crashed at run's 22 and 43. This is shown on the graphs where the point plotted is at 0 for the Y-axis. The exception thrown by the system was :

Exception during Thread Dispatch: org.omg.CORBA.NO_MEMORY

The server application had to be shut down and restarted. The cause of the problem seems to be that after the supplier and consumer applications had finished, the server did not destroy the objects it had used.

The ORBacus system also crashed at run 33, this time there was no diagnostic information printed to the screen. The supplier, consumer and server froze, they had to be shut down and restarted again.

The iBus supplier and consumer was much faster than the other for connecting to an Event Channel. This is not surprising, since the CORBA Event Service requires multiple steps to establish a connection, whereas the iBus only required a single method call. TCDEvents was the fastest for the consumer connection compared with the other CORBA models; jEVENTS and ORBacus. The supplier connection times of TCDEvents and jEVENTS were practically the same, with ORBacus being the slowest. TCDEvents uses the Naming Service to obtain an IOR reference to the Event Channel, this is the CORBA standard way of locating IORs. Whereas jEVENTS and ORBacus Event Channel references were stored in a file which meant that every time a consumer and supplier wanted to connect to an Event Channel, it had to open the file and read the IOR. This operation caused the slowness compared to TCDEvents.

With the sending and receiving of 500 events, TCDEvents was the fastest, with the other two CORBA models, jEVENTS and ORBacus being a few seconds slower. The non CORBA model, iBus was considerably slower, some 30 seconds. It is very difficult to determine why this is without having access to the source code. It could be, that the operations to push the event onto the channel and then to push it onto the consumers may have a sleep operation for a few milliseconds. If this is the case, then this would definitely account for the performance difference.

The Pull Model Test

This consisted of the same type of tests as the push model.

- **Connecting a Pull Consumer to an Event Channel**

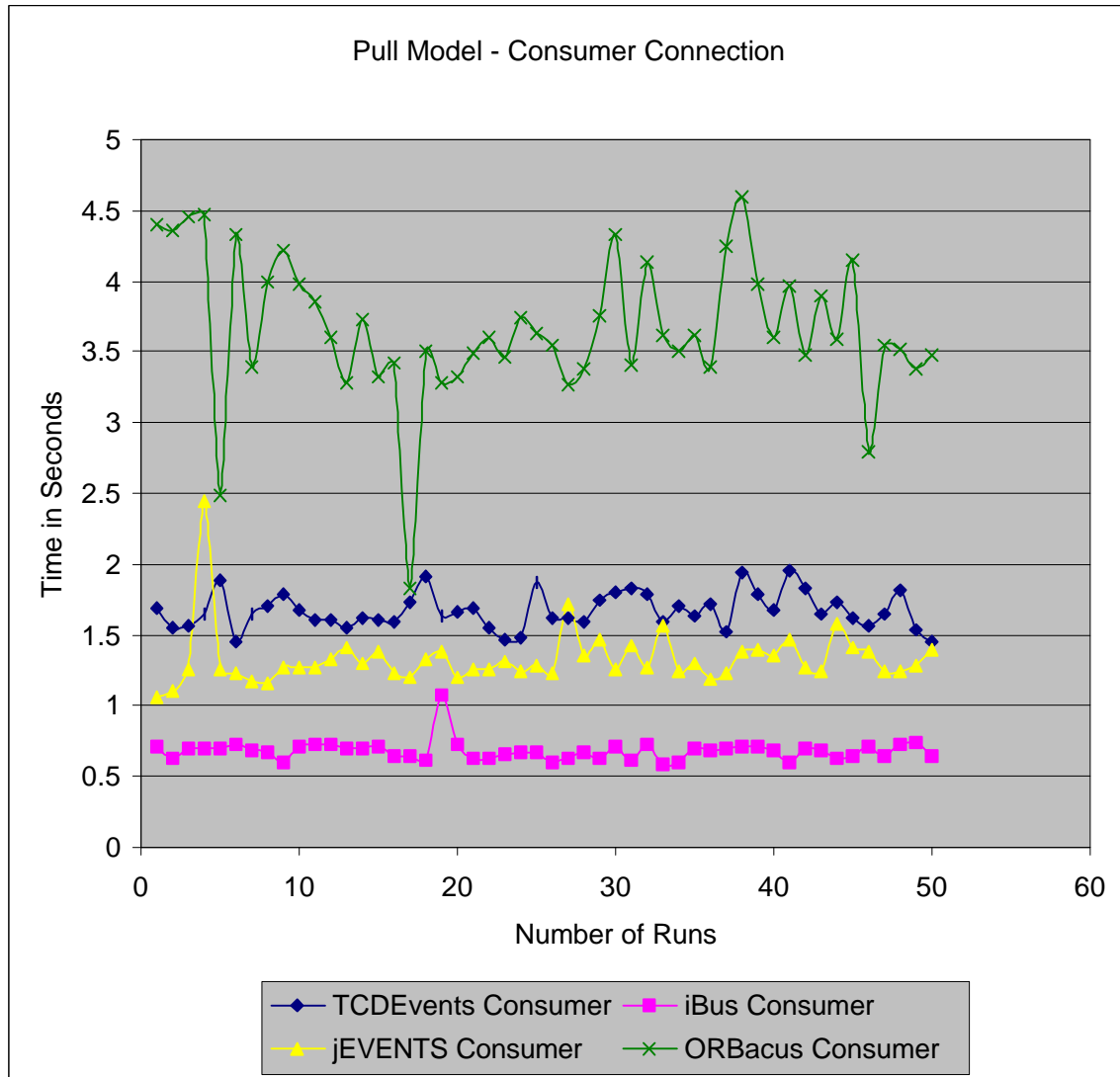


Figure 6-5 Connection of a Pull Consumer

Figure 6-5 shows similar results as the push model. The iBus event model was the fastest and the ORBacus the slowest to connect. The connection times for TCDEvents and jEVENTS were very similar considering the recordings have a fault tolerance of +/- 5%.

- **Connecting a Pull Supplier to an Event Channel**

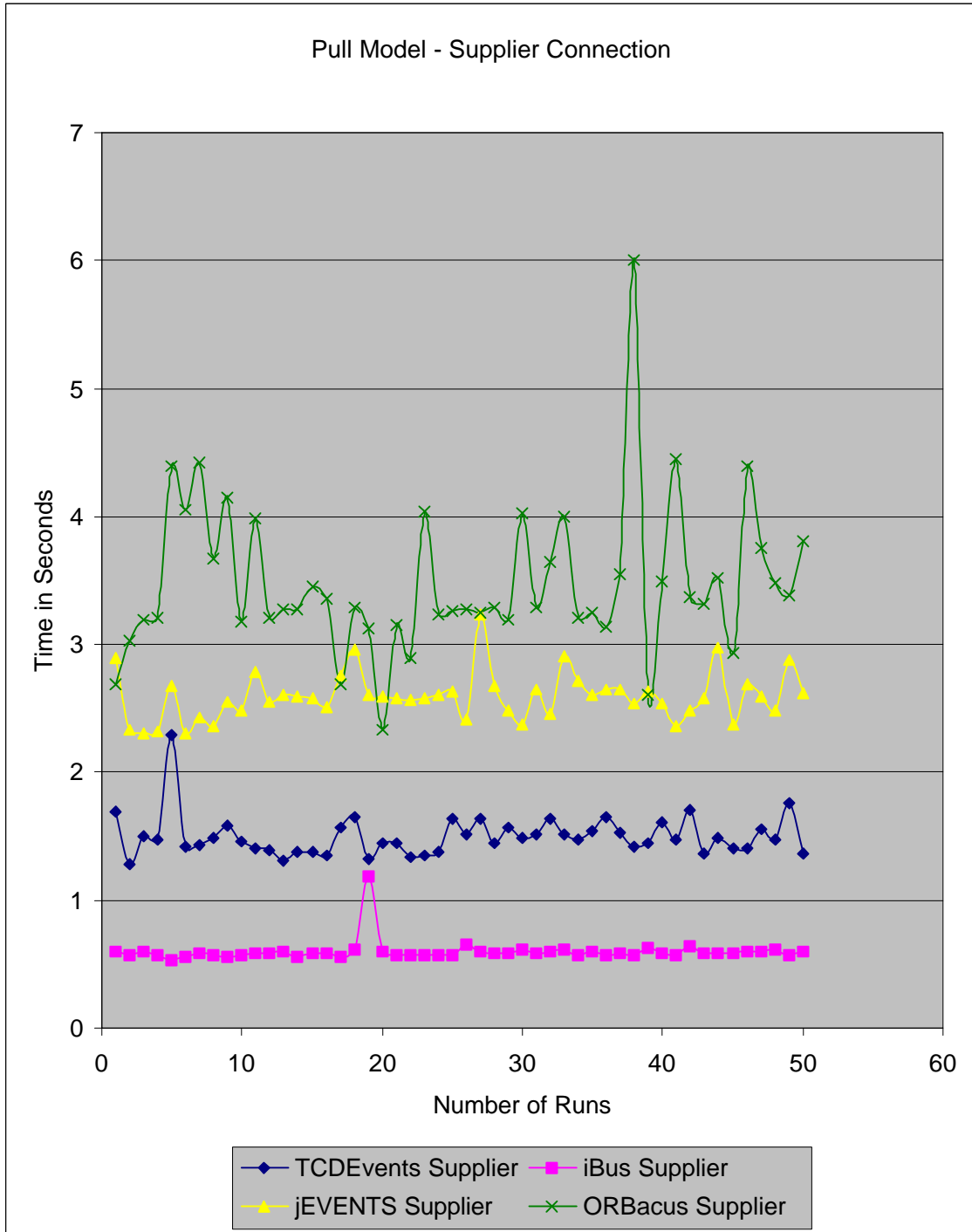


Figure 6-6 Connection of a Pull Supplier

Figure 6-6 again shows very similar results as the push model, with TCDEvents being the fastest CORBA Event Service.

- Pull Supplier sending 100 Events

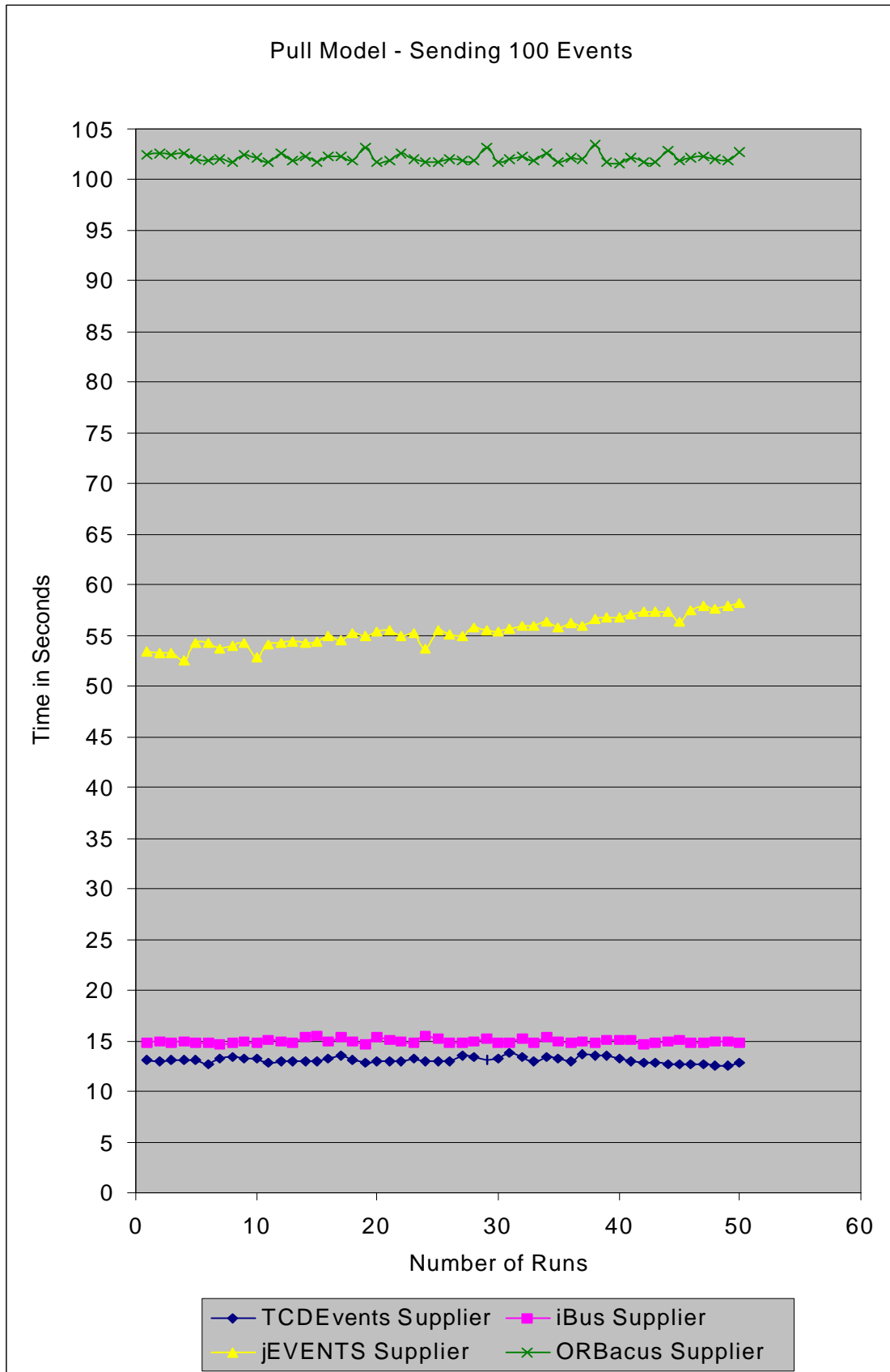


Figure 6-7 Pull Supplier sending 100 Events

- **Pull Consumer receiving 100 Events**

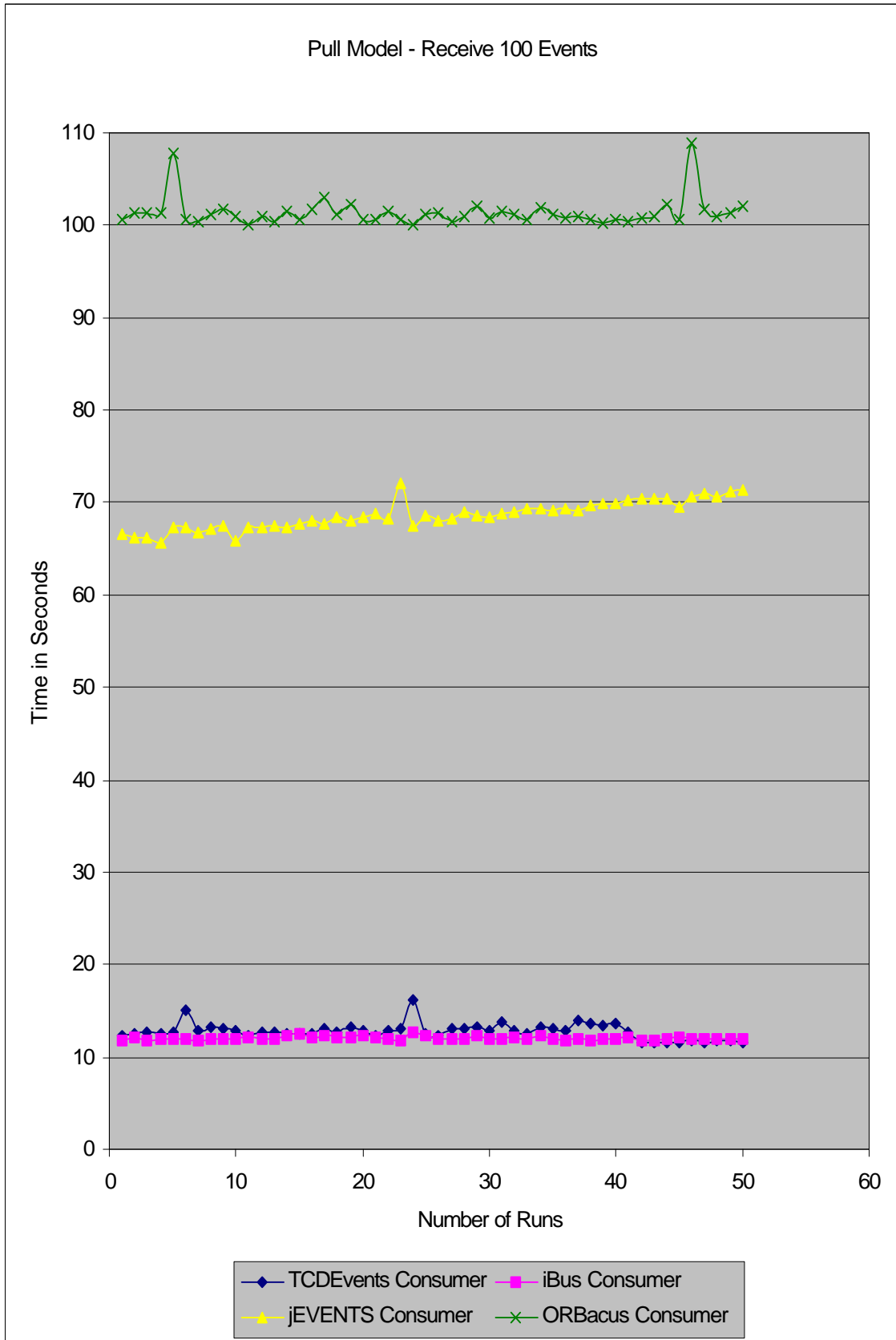


Figure 6-8 Pull Consumer receiving 100 Events

The results for sending and receiving 100 events as shown in Figures 6-7 and 6-8 are very similar. TCDEvents is considerably faster than the other CORBA Event Services.

Average Times for Pull Model Tests

Average time in Seconds Over 50 Runs				
	Connect Consumer	Connect Supplier	Send 100 Events	Receive 100 Events
TCDEvents	1.67164	1.49886	13.08058	12.7766
jEVENTS	1.33004	2.59468	55.41898	68.63788
ORBacus	3.66652	3.48348	102.17154	101.38274
iBus	0.68062	0.59996	14.96726	12.02412

Table 6-2 Average test times for the Pull model

Pull Model Test Conclusions

The connection times for the supplier and consumers followed the same pattern as the push model for the same reasons. None of the event models crashed during the testing, unlike they did for the push model. This could be due to the fact that the test runs were only dealing with 100 events instead of 500.

With the sending and receiving of 100 events, TCDEvents and iBus were almost identical. Whereas jEVENTS and ORBacus were considerably slower. With ORBacus there was a sleep method call of a few hundred milliseconds in the *pull()* method. As for jEVENTS it is difficult to conclude why it was so slow. There was a very large

period of time, approximately 20 seconds, between the time the supplier and consumers were connected to the Event Channel and the time the consumer received its first event. By reducing this period, so that the event appears at the consumer almost immediately, would greatly improve jEVENTS performance. Without having access to the source code it is very difficult to pin point the actual bottleneck.

Performance of the Push Model

The push style communication model is the most commonly used in distributed systems. This is why it is very important that it performs efficiently. TCDEvents has implemented a thread pool at the consumer side of the Event Channel to increase delivery performance of the event data to the consumers.

To measure the performance of delivering 500 events, the number of threads in the thread pool was incremented by one in order to determine the optimal number of threads that are required for good performance. Figure 6-9 shows the performance measurements. Considering there is a +/- 5% fault tolerance with the readings, the results show that the push model is most efficient when there are between 2 and 14 threads in the thread pool. In the measurements taken, the fastest time was recorded when there were 12 threads in the pool.

Increased performance could be achieved by compiling the TCDEvents source code with a JIT (Just-In-Time) compiler, which is available in JDK 1.1.6.

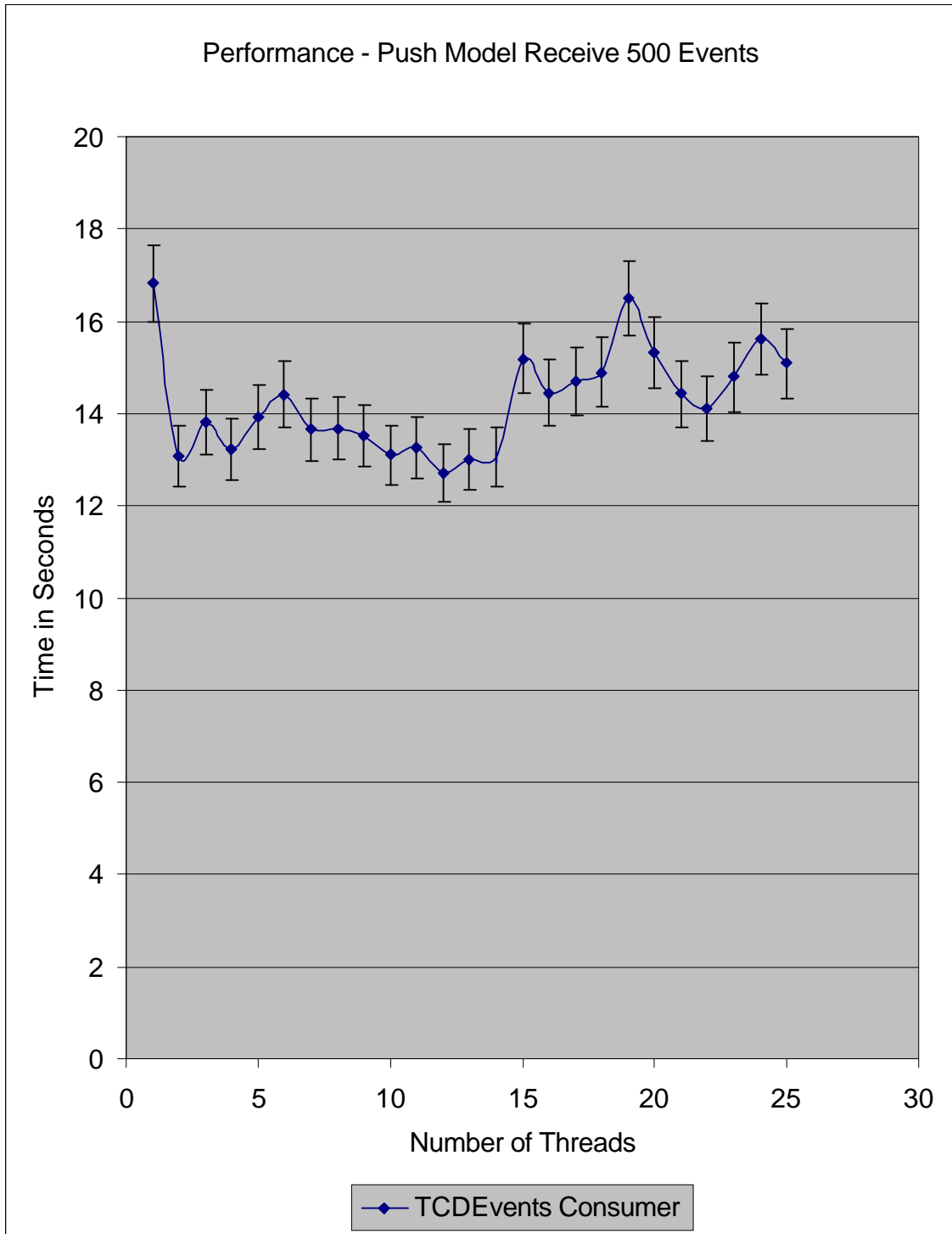


Figure 6-9 Performance Measurements for the Push Model

Summary

TCDEvents was the easiest event model to use, mainly due to the facts that a server application was provided and a GUI wizard application generated supplier and consumer applications at a push of a button. It also uses the OMG standard way of retrieving IOR's by making use of the Naming Service. The other CORBA models, jEVENTS and ORBacus did not provide a server application. This provides an extra burden on the software developer. The Event Channel IOR's were stored in a file, which adds extra restrictions, especially if either the supplier or the consumer application is on another host to the server, then it will need the correct permissions in order to access the file. iBus did not need a server as it uses URL's as Event Channels, which means that applications did not have to be registered with an Implementation Repository. All event models were interoperable. iBus could be used with an IIOP bridge to communicate with CORBA applications. Another advantage with TCDEvents was that the server could contain multiple Event Channels, the other event models could only have a single Event Channel. On the whole, TCDEvents performed better than the other CORBA models in terms of connecting suppliers/consumers and the sending/receiving of events. Only the iBus outperformed TCDEvents when connecting applications to the Event Channel. This is not surprising as one of the drawbacks to the CORBA Event Service is the multiple steps required to connect an application to the channel.

7

Conclusion

There has been increasing interest for distributed technologies which has led to the emergence of several object oriented distributed middleware solutions. One of the most common in use today is the Common Object Request Broker Architecture (CORBA), specified by the Object Management Group (OMG). CORBA provides an object orientated infrastructure that allows objects to communicate with each other, regardless of the programming language used to implement the objects and the hardware platforms that they run on. At the heart of CORBA is the Object Request Broker (ORB) that is basically a communication bus that the objects use to pass messages to each other. The OMG has also specified a set of services that are the basic building blocks for a distributed object infrastructure. These services provide the next step up in the evolutionary chain towards creating distributed components, that are commonly referred to as Business Objects.

One of the basic requirements in distributed systems is for objects to communicate events to each other in an asynchronous manner. However, the basic communication paradigm for CORBA is synchronous. This means that if a client makes a request to a server object, it must wait until the server has processed the request and returned the result back to the client, before the client can continue. In many situations, this synchronous communication between client and server is unsuitable, for this reason the OMG have specified a CORBA Event Service.

The CORBA Event Service decouples the communication between clients (suppliers) and servers (consumers) through Event Channels. Suppliers generate events and send them to consumers via the Event Channel without knowing the identity of the consumer or how many consumers are receiving the event. Similarly, consumers receive and process the data from suppliers via the Event Channel without knowing the identity of the supplier. Event Channels are standard CORBA objects and

communication between suppliers and consumers is achieved using the standard CORBA requests. There are two basic communication models associated with the Event Service. The Push model allows a supplier of an event to initiate the transfer of event data to consumers. The other is the Pull model, this allows consumers of events to request event data from a supplier. It is also possible to mix the communication styles. A Push-Pull model is where there are push suppliers and pull consumers and a Pull-Push model, consists of pull suppliers and push consumers. The most commonly used communication style is the Push model. The CORBA Event Service is suitable for many application domains, because of its general architecture model. However, there are many important features that the Event Service lacks which makes it unsuitable for application domains such as telecommunications and real-time. Event filtering and correlation is missing from the specification, events are delivered to all consumers that are connected to the Event Channel whether they are interested in the event or not. This means that each consumer must filter the events to find the ones it is interested in. For event correlation, some consumers can only execute when events arrive from a particular set of suppliers. Again it is the responsibility of consumer applications to perform the correlation of events. This is very costly in terms of work load, which can be particularly problematic if consumers are run on low powered machines. Other problems include the lack of a persistent Event Channel, if the Event Channel object fails then the event information contained within the channel is lost. The lack of Quality of Service is also a problem, some implementations may provide guaranteed delivery semantics while some other implementations may only provide at-least-once semantics. Depending on the delivery semantics that are required for an application domain, dictates which vendors Event Service implementation to choose. There is also no federation associated with the Event Service, an event server in one local organization has no way to contact an event server in another local organization. For these reasons the OMG have proposed a new service called the Notification Service. This service will address issues such as Event Filtering, Event Priority and Persistent Event Channel. It does not go all the way to solve all the problems of the Event Service, but it will be suitable for telecommunications domains. The next generation of CORBA, which will contain a Messaging Service will resolve the remaining issues. The Messaging Service is based upon the Message Oriented Middleware technology. The integration of Message Oriented Middleware and CORBA will make the CORBA model more flexible and time tolerant.

The CORBA Time Service makes extensive use of the CORBA Event Service Push model, for events that have to be delivered at specific times.

There have been many implementations of event models, both, CORBA and non-CORBA, from commercial vendors and academic institutions. This project investigated a variety of different event models. Most of the CORBA commercial implementations, with the exception of DIAS from ICL, do not add their own functionality to overcome some of the drawbacks. The problem of adding non-compliant OMG functionality is that the Event Service is not interoperable with other ORB's. The DIAS event model is a multicast service. The OMG has specified that an Event Service is a broadcast model, a consumer will receive every event. The purpose of the DIAS multicast is to satisfy the requirements of the customer which required a large volume of messages to be communicated in a short period of time. DIAS is not interoperable with other ORB's, since data is transferred over UDP and not IIOP. Academic institutions have developed event models based on the CORBA Event Service but have implemented their own solutions to the various drawbacks that the CORBA service has. The TAO ORB developed at Washington University, consists of a real-time Event Service that arguments the CORBA Event Service model by providing source-based and type-based filtering, event correlation and real-time dispatching. This ORB is currently deployed at McDonnell Douglas in St. Louis, where it is being used to develop operation flight programs for the next generation avionics systems. Another academic event model is COBEA which has been developed at Cambridge University. This system is an extension of the CORBA Event Service that supports parameterized filtering, fault-tolerance, access control and composite events, all of which are not in the OMG Event Service specification. The filtering and correlation mechanisms of the COBEA system are more powerful than the TAO Event Service as it is not limited to placing filters at the Event Channel. The non-CORBA models are based on similar principles and the majority can be used with CORBA applications by using an IIOP bridge. Two Java event models are InfoBus from Sun Microsystems that allows Java Bean components to communicate with each other, and iBus middleware. iBus supports intranet applications such as content delivery systems, groupware, fault-tolerant client-server systems, and multimedia applications. It provides multicast channels that allow Java applications to interact using the Push or Pull communication model. It seems that currently there is

not one event model that fits all application domains, the best way to select a suitable event model is to look at the domain and find the model that matches best.

TCDEvents is a Java implementation of the CORBA Event Service. TCDEvents is developed using an incremental development model, that consisted of defining the requirements of the system, producing a detailed design and testing to ensure the system meets the requirements. One of the major requirements of TCDEvents is that it is easy to use and removes much of the burden from the software developer by providing an Event Server and a GUI wizard. The Event Server contains a Channel Manager object that allows the software developer to create and manipulate the Event Channels. The Channel Manager is not defined by the CORBA Event Service specification, it is extra functionality to help the developer to control the Event Channels of the server. It has an IDL definition and is implemented according to the CORBA standard thus not effecting any interoperability issues. The Event Server can contain multiple Event Channels. The number of channels within a single server is restricted only by the hardware specifications. The design made extensive use of the Unified Modeling Language (UML), and designed independent of any programming language. An Event Server application is provided, registering the server with the Implementation Repository creates the Event Channels that are specified at the command line and creates the Channel Manager object. The Event Channels and the Channel Manager names are registered with the Naming Service. Using the Naming Service provides the CORBA compliant way for applications to obtain IOR's. The supplier and the consumer applications contact the Naming Service to resolve the channel name in order to connect to the channel. The Push model makes extensive use of Threads and has a thread pool associated with the consumer end of the Event Channel. The purpose of using threads is to increase the delivery performance of event data to consumer applications. The actual delivery semantics are based on best-effort. The Event Channel delivers event data at least once to every consumer application. The Pull model architecture is slightly different, it does not use thread pools. Instead the proxy consumer object that is created when a supplier connects to the Event Channel, creates its own thread that is responsible for asking for events. The GUI wizard application is used to automatically generate supplier and consumer applications. Testing was carried out on a class to class basis. Once a class was completed a test harness was written to test the functionality of the class to ensure it

was functioning correctly. When the whole coding of TCDEvents was completed, some demonstration programs were written to test the functionality of the Channel Manager object and the various combination of communication styles.

The final part of the project evaluated TCDEvents with three other Java event models. Two CORBA models; jEVENTS and ORBacus, and one non-CORBA model iBus were used for the evaluation. TCDEvents is the easiest event model to use, mainly due to the fact that a server application is provided and also the GUI wizard generates supplier and consumer applications at a push of a button. The other CORBA models, jEVENTS and ORBacus do not provide a server application, which require extra coding burden to the software developer. Their Event Channel IOR's are stored in a file and this adds extra restrictions, especially if either the supplier or consumer application are on different host from the server. The applications will need the correct permissions in order to access the file and read the IOR. iBus did not need a server as it uses URL's as Event Channels, this means that applications do not have to be registered with an Implementation Repository. All event models are interoperable, iBus can be used with an IIOP bridge to communicate with CORBA applications. Another advantage with TCDEvents is that the server could contain multiple Event Channels, the other event models can only have a single channel. On the whole, TCDEvents performed better than the other CORBA models in terms of connecting suppliers/consumers and the sending/receiving of events. Only the iBus out performed TCDEvents when connecting applications to the Event Channel. This is not surprising as one of the drawbacks to the CORBA Event Service are the multiple steps required to connect an application to the channel.

8

Glossary

ACRONYMS

AI	Artificial Intelligence
CORBA	Common Object Request Broker
COSS	Common Object Services Specification
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DII	Dynamic Invocation Interface
IDL	Interface Definition Language
IP	Internet Protocol
ISDN	Integrated Services Digital Network
IT	Information Technology
JVM	Java Virtual Machine
MOM	Message Oriented Middleware
OMG	Object Management Group
ORB	Object Request Broker
RMI	Remote Method Invocation
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
TINA	Telecommunication Information Network Architecture
TMN	Telecommunication Management Network
UDP	User Datagram Protocol
UML	Unified Modeling Language
UTC	Universal Time Coordinated

9

References

- BN:84** Xerox Palo Alto Research Center, Andrew Birrell and Bruce Nelson
“Implementing Remote Procedure Calls”, ACM Transactions on
Computer Systems, Vol2, No.1, February 1984, Pages 39-59
- CMJB:98** Chaoying Ma and Jean Bacon. *“COBEA – A CORBA-Based Event
Architecture”*. University of Cambridge, April 1998.
- CM:95** Tina Consortium, Martin Chapman and Stefano Montesi, *“Overall
Concepts and Principles of TINA”*, Version 1.0 17th February 1995.
<http://www.tinac.com/>
- DSSV8:96** Douglas C. Schmidt, Steve Vinoski *“Object Interconnections:
Distributed Callbacks and Decoupled Communication in CORBA
(Column 8)”* , October 1996 Issue of SIGS C++ Report Magazine.
- DSSV9:97** Douglas C. Schmidt, Steve Vinoski *“Object Interconnections: The
OMG Events Service (Column 9)”* , February 1997 Issue of SIGS C++
Report Magazine.
- DSSV10:97** Douglas C. Schmidt, Steve Vinoski *“Object Interconnections:
Overcoming Drawbacks in the OMG Events Service (Column 10)”* ,
June 1997 Issue of SIGS C++ Report Magazine.
- Elvin:97** Bill Segall and David Arnold. *“Elvin has left the building: A
publish/subscribe notification service with quenching”*. University of
Queensland, 1997.

-
- FH:96** Fred Halshall “*Data Communications, Computer Networks and Open Systems*”, Fourth Edition, Addison-Wesley, 1996.
- FS:97** Broadcom Eireann Research, Fergal Somers, “*HYBRID: Intelligent Agents for Distributed ATM Network Management*”, 1997.
- GB:98** Gerald Brose “*JacORB Programming Guide, v0.9*”. University of Berlin, Germany, March 18th 1998.
- HLS:97** Timothy H. Harrison, David L. Levine and Douglas C. Schmidt “*The Design and Performance of a Real-time Event Service*”. OOPSLA 1997 (Atlanta, GA), ACM, October 1997.
<http://www.cs.wustl.edu/~schmidt/oopsla.html>
- iBus:97** Silvano Maffei “*iBus – The Java Intranet Software Bus*”. SoftWired AG, February 1997.
- ICL:98** ICL Object Software Laboratories. “*DAIS Multicast Event Service*” White Paper, January 1998.
- IONA:96** IONA “*OrbixTalk – The White Paper*” Technical Report. IONA Technologies PLC, April 1996.
- IONA:97** IONA “*OrbixEvents Programmer’s Guide*”. IONA Technologies PLC, December 1997.
- JB:95** John Bates, University of Cambridge, “*Presentation Support for Distributed Multimedia Applications*”, 1995.
- JL:97** Professor James Landay, Berkeley University, “*Model View Controller*” November 26, 1997.
<http://bmrc.berkeley.edu/courseware/cs160/fall97/lectures/11-26-97/>

-
- MOM:98** “*Message Oriented Middleware Association*”
<http://www.moma-inc.org/>
- OHE:96** Robert Orfali, Dan Harkey and Jeri Edwards, “*The Essential Client/Server Survival Guide*” Second Edition,
Publisher: John Wiley and Sons, INC. 1996.
- OHE:97** Robert Orfali, Dan Harkey and Jeri Edwards, “*Instant CORBA*”
Publisher: John Wiley and Sons, INC. 1997.
- OMG:93** Object Management Group, “*Object Management Architecture Guide*”. John Wiley and Sons, Inc., 1993
- OMG:95a** Object Management Group, “*The Common Object Request Broker : Architecture and Specification*” , 2.0 ed., July1995.
- OMG:95b** Object Management Group, “*CORBAServices: Common Object Services Specification*” , Revised Edition, 95-3-31 ed, March. 1995.
<ftp://www.omg.org/pub/docs/formal/97-07-13.pdf>
- OMG:95c** Object Management Group, “*Common Facilities Architecture Document*” Revision 4.0, November 1995.
- OMG:95d** Object Management Group, “*Common Facilities Roadmap*”
January 1995.
- OMG:96e** Object Management Group, “*OMG Messaging Service*” RFP. OMG
Document No. ORBOS/96-03-16, March 1996.
- OMG:96f** Object Management Group, “*OMG Notification Service*” RFP. OMG
Document No. Telecom/96-11-03, November 1996.

- OOC:98** Object-Orientated Concepts Inc , “*ORBacus for C++ and Java*”.
Versions 3.0, 1998.
<http://www.ooc.com/ob/>
- OUT:97** OUTBACK “*jEVENTS – Java-based Event Service User’s Guide*”.
OutBack Resource Group Inc. 1997
- RIR:96** Ron I. Resnick, “*CORBA Async Requirements*” 1996
- SCT:95** Gradimir Starovic, Vinny Cahill and Brendan Tangney. “*An Event Based Object Model for Distributed Programming*” Trinity College, Dublin.
- VS** NEC America Inc., Veli Sahin, “*Telecommunications Management Network Principles, Models and Applications*”.

Appendix A

TCDEvents IDL Definitions

*This appendix lists the IDL definitions in the CORBA Event Service **CosEventComm** and **CosEventChannelAdmin** modules as specified by the CORBA standard. It also lists the Event Channel administrator interface **EventServerAdmin** which is not specified by the CORBA standard. This interface allows the creation and manipulation of Event Channels within the TCDEvents Server.*

CosEventComm.idl

```
#ifndef __COSEVENTCOMM_IDL
#define __COSEVENTCOMM_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();
    };
};

#endif
```

CosEventChannelAdmin.idl

```
#ifndef __COSEVENTCHANNELADMIN_IDL
#define __COSEVENTCHANNELADMIN_IDL

#include "CosEventComm.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::PullSupplier 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();
        void setChannelCapacity(in long capacity);
        long getChannelCapacity();
    };
};

#endif
```

EventServerAdmin.idl

```
#ifndef __EVENTSERVERADMIN_IDL
#define __EVENTSERVERADMIN_IDL

#include "CosEventChannelAdmin.idl"

module EventServerAdmin
{
    exception duplicateChannel{};
    exception noSuchChannel{};

    interface ChannelManager
    {
        typedef sequence<string> channelSeq;

        CosEventChannelAdmin::EventChannel createChannel(in string channel_name)
            raises(duplicateChannel);

        CosEventChannelAdmin::EventChannel findChannel(in string channel_name)
            raises(noSuchChannel);

        void destroyChannel(in string channel_name)
            raises(noSuchChannel);

        channelSeq listAllChannels();

        string findChannelName(in CosEventChannelAdmin::EventChannel channel_ref)
            raises(noSuchChannel);
    };
};
#endif
```

Appendix B

TCDEvents Source Code

This appendix lists the source code for both, the implementation of TCDEvents and the GUI Wizard application

TCDEvents Source code

```
//-----
/*
CLASS Argument
*/
//-----

package TCDEvents.EventsImpl;

/**
 *
 * This class handles the command line arguments, for the
 * Event Server application.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class Argument
{
    private String [] args;
    private String [] channel_id;
    private int numberOfChannels;
    private String serverName;
    private boolean rebind_flag;
    private int bufferSize;
    private int timeOut;

    /**
     *
     * Does the initial argument check that arguments has been specified.
     *
     * @param args A string array that contains the command-line arguments.
     */

    public Argument(String args[])
    {
        this.args = args;

        // create array to store the channel ID's using
        // the argument lenght to be the maximum.
        // The actual array of strings are created later.
        channel_id = new String[args.length];
        numberOfChannels = 0;

        rebind_flag = false;
        bufferSize = 0;
        timeOut = 0;

        if (args == null || args.length < 1)
        {
            usage();
        }

        // Check to make sure that there is a channel ID as the
        // first argument, if not display usage and exit.
        check_for_channel();

    } // End of Argument

    /**
     *
     * Stores the channel name in the channel ID array and increments
     * the channel counter.
     *
     * @param index The index into the channel ID array.
     */

    public void determine_channel(int index)
    {
        if ( !check_switches(index) )
        {
            channel_id[index] = new String(args[index]);
            numberOfChannels++;
        }
    }

    } // End of determine_channel

```

```
/**
 *
 * Determines if the rebind switch has been specified. If so then
 * a flag is set for later reference.
 *
 * @param index The index into the command-line argument array.
 *
 */
public void determine_rebind(int index)
{
    if ( args[index].equals("-rebind") )
    { // -rebind switch specified, so set the flag.
        rebind_flag = true;
    }
} // End of determine_rebind

/**
 *
 * Determines if the server name command-line argument is specified.
 * If so the name of the server is stored for future reference.
 * A boolean value is returned indicating if the server name switch
 * has been specified or not.
 *
 * @param index The index into the command-line argument array.
 *
 */
public boolean determine_serverName(int index)
{
    boolean status = false;

    if ( args[index].equals("-server_name") )
    {
        // move to the next argument which is the name.
        if ( index < (args.length-1) )
        {
            index++;
        }
        else
        {
            usage();
        }

        // check to ensure we don't have a switch command.
        // never trust the user input !!
        if ( check_switches(index) )
        {
            usage();
        }

        serverName = new String(args[index]);
        status = true;
    }

    return status;
} // End of determine_serverName

/**
 *
 * Determines if the buffer command-line argument is specified.
 * If so the size of the buffer is stored for future reference.
 * If a value of less than or equal to 0 is specified, the size
 * takes the default value, 1000. A boolean value is returned
 * indicating if the buffer switch has been specified or not.
 *
 * @param index The index into the command-line argument array.
 *
 */
public boolean determine_buffer(int index)
{
    boolean status = false;

    if ( args[index].equals("-buffer") )
    {
        // move to the next argument which is the buffer size
        if ( index < (args.length-1) )
        {
            index++;
        }
        else
        {
            usage();
        }
    }
}
```

```

    }

    // check to ensure we don't have a switch command.
    // never trust the user input !!
    if ( check_switches(index) )
    {
        usage();
    }

    Integer size = new Integer(args[index]);
    bufferSize = size.intValue();
    status = true;
}
return status;

} // End of determine_buffer

/**
 *
 * Determines if the timeout command-line argument is specified.
 * If it is then the server will time-out after the specified
 * number of minutes. Otherwise the default value is 10 minutes.
 *
 * @param index The index into the command-line argument array.
 *
 */
public boolean determine_timeout(int index)
{
    boolean status = false;

    if ( args[index].equals("-timeout") )
    {
        // move to the next argument which is the buffer size
        if (index < (args.length-1) )
        {
            index++;
        }
        else
        {
            usage();
        }

        // check to ensure we don't have a switch command.
        // never trust the user input !!
        if ( check_switches(index) )
        {
            usage();
        }

        Integer value = new Integer(args[index]);
        timeOut = value.intValue();
        status = true;
    }
    return status;
} // End of determine_timeout

/**
 *
 * Returns the channel name ID array.
 * This array contains all the unique channel names that are
 * associated with the Event Server.
 *
 * @return channel_id An array that contains all the channel names.
 *
 */
public String [] channel_names()
{
    return channel_id;
} // End of channel_names

/**
 *
 * Returns the number of channels that the Server application has.
 *
 * @return numberOfChannels The number of channels associated with the server.
 *
 */
public int number_of_channels()
{
    return numberOfChannels;
} // End of number_of_channels

```



```
/**
 *
 * Returns the name of the Server.
 *
 * @return server_name The name of the Server.
 *
 */
public String server_name()
{
    return serverName;
} // End of server_name

/**
 *
 * Returns the status of the rebind switch. If it is specified
 * at the command-line then the method returns true, otherwise
 * false is returned.
 *
 * @return rebind_flag True if rebind has been specified, otherwise false
 *
 */
public boolean rebind()
{
    return rebind_flag;
} // End of rebind

/**
 *
 * Returns the size of the buffer associated with the channel.
 *
 * @return bufferSize The size of the channels buffer.
 *
 */
public int buffer_size()
{
    return bufferSize;
} // End of buffer_size

/**
 *
 * Returns time in minutes associated with the Event Channel timeout.
 *
 * @return timeOut The value in minutes of the Server timeout.
 *
 */
public int timeout_value()
{
    return timeOut;
} // End of timeout_value

/**
 *
 * Ensures that there is at least the mandatory channel name.
 * Otherwise the usage message is displayed.
 *
 */
private void check_for_channel()
{
    if ( check_switches(0) )
    {
        usage();
    }
} // End of check_for_channel

/**
 *
 * Checks for the optional command-line switches. If they exist
 * then the method returns true, otherwise false is returned.
 *
 * @param index The index into the command-line argument array.
 * @return True if only a channel name specified, otherwise false.
 *
 */
private boolean check_switches(int index)
{
```

```

// check for the presence of the switch commands.
if ( args[index].equals("-rebind") || args[index].equals("-server_name")
    || args[index].equals("-buffer") || args[index].equals("-timeout"))
{
    return true;
}
else
{
    return false;
}
} // End of check_switches

/**
 *
 * Displays the usage of the Server application, then exits.
 *
 */

private void usage()
{
    System.err.println("");
    System.err.println("Usage: EventServer <channel_id> [-rebind] [-server_name name] [-buffer size] [-timeout value]");
    System.err.println("");
    System.err.println(" channel_id : Mandatory, unique name of the Event Channel");
    System.err.println(" -rebind   : Used to override the original Name binding");
    System.err.println(" -server_name : Name of the server registered with the ImpRep, default ES");
    System.err.println(" -buffer   : Buffer size of the Event Channel");
    System.err.println(" -timeout  : Value in minutes of the Server timeout");
    System.exit(1);
} // End of usage
} // End of class Argument

//-----
/*
CLASS BoundedQueue
*/
//-----

package TCDEvents.EventsImpl;

import java.util.*;

/**
 *
 * This is the bounded queue class. It allows event data to be
 * added to the queue and removed from the queue. The default
 * size of the queue is 1000.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

public class BoundedQueue
{
    protected Object theArray[];
    protected int front = 0;
    protected int back = -1;
    protected int size = 0;
    protected int eventsPending = 0;

    static final int DEFAULT_CAPACITY = 1000;

    /**
     *
     * Creates the queue with the default size of 1000.
     *
     */

    public BoundedQueue()
    {
        // creates the array of objects with the default
        // buffer size.
        size = DEFAULT_CAPACITY;
        theArray = new Object[DEFAULT_CAPACITY];
        back = size - 1;
    } // End of BoundedQueue

```

```
/**
 *
 * Creates the queue with the size of the value specified.
 *
 * @param The size of the queue to be created.
 *
 */
public BoundedQueue(int size)
{
    if ( size > 0 )
    {
        this.size = size;
        theArray = new Object[size];
        back = size - 1;
    }
} // End of BoundedQueue

/**
 *
 * Determines if there are any events in the queue pending delivery.
 *
 * @return True if the queue is empty, otherwise false.
 *
 */
public boolean isEmpty()
{
    return (eventsPending == 0);
} // End of isEmpty

/**
 *
 * Determines if the queue is full.
 *
 * @return True if the queue is full, otherwise false.
 *
 */
public boolean isFull()
{
    return (eventsPending == size);
} // End of isFull

/**
 *
 * The number of events that are pending delivery.
 *
 * @return The number of events in the queue.
 *
 */
public int eventsPending()
{
    return eventsPending;
} // End of eventsPending

/**
 *
 * The size of the queue associate with the Event Channel.
 *
 * @return The size of the queue.
 *
 */
public int getCapacity()
{
    return size;
} // End of getCapacity

/**
 *
 * Extends the size of the Event Channel queue.
 *
 * If the value of capacity is less than the number
 * of events that are pending delivery, then the
 * queue size will be set to the number of events
 * that are pending. This way it will ensure that
 * no events are lost when reducing the size of the
 * queue.
 *
 */
```

```

* @param capacity The new size of the queue.
*
*/

public void setCapacity(int capacity)
{
    Object [] newArray;

    // make sure the new queue size is not less than the
    // number of events waiting delivery.
    if (capacity < eventsPending)
    {
        capacity = eventsPending;
    }

    newArray = new Object[capacity];

    // copy elements that are logically in the queue
    for (int i=0; i<eventsPending; i++, front = increment(front))
    {
        newArray[i] = theArray[front];
    }

    theArray = newArray;
    front = 0;
    back = eventsPending -1;
    size = capacity;
} // End of setCapacity

/**
*
* Places an event data item into the back of the queue.
* Once the data item has been added the event pending count
* is incremented.
*
* @param eventData The event data item object.
*
*/

public void putEventItem(Object eventData)
{
    if (eventData != null && !isFull())
    {
        back++;
        if (back >= size)
        {
            back = 0;
        }
        theArray[back] = eventData;
        eventsPending++;
    }
} // End of putEventItem

/**
*
* Removes the event data item from the front of the queue.
* The item removed from the queue is ready for delivery
* to all the connected consumers. Once the data item has
* been removed then the event pending count is decremented.
*
* @return The event data item that has been removed.
*
*/

public Object takeEventItem()
{
    Object eventData = null;
    if (!isEmpty())
    {
        eventData = theArray[front];
        theArray[front] = null;
        front++;
        if (front >= size)
        {
            front = 0;
        }
        eventsPending--;
    }
    return eventData;
} // End of takeEventItem

```

```

/**
 *
 * Method to increment with wraparound.
 *
 * @param index The index into the queue range.
 * @return index+1, or 0 if index is at end of queue.
 *
 */
private int increment(int index)
{
    if (++index == theArray.length)
    {
        index = 0;
    }
    return index;
} // End of increment
} // End of class BoundedQueue

//-----
/**
 * CLASS Channel extends LocateNamingService
 */
//-----

package TCDEvents.EventsImpl;

import java.util.*;

import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

import TCDEvents.CosEventChannelAdmin.*;

/**
 *
 * This class is used as a helper to the Event Channels that are
 * registered with the naming service. It is used to determine
 * either the context name of the event channel or its object
 * reference.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

public class Channel extends LocateNamingService
{
    private ORB orb;
    private NamingContext rootContext = null;
    private NamingContext channelContext = null;

    private Vector eventChannelArray;
    private Vector push_consumers;
    private Vector pull_consumers;
    private Vector push_suppliers;
    private Vector pull_suppliers;

    private ResolveNames resolveChannel;

    private int failChannelNumber = 0;

    static final String nameEC = "eventChannel";

    /**
     *
     * Class constructor, resolves the Naming Service and it's
     * root context
     *
     * @param orb The Object Request Broker the Event Channel is connected to.
     *
     */
    public Channel(ORB orb)
    {
        // create the event channel vector array object.
        eventChannelArray = new Vector();

        // get the root context of the naming service
        rootContext = getRootContext();

        // create the resolveNames object to resolve the channel name.
        resolveChannel = new ResolveNames();
    }

```

```

// initialise the orb object
this.orb = orb;

// initialise the supplier/consumer vector objects.
push_consumers = new Vector();
pull_consumers = new Vector();
push_suppliers = new Vector();
pull_suppliers = new Vector();

} // End of EventChannel

/**
 *
 * Adds the Channel Name that has been specified by the command
 * line arguments to the Naming Service. If there is only one
 * event channel for the server, and an error was generated
 * when contacting the Naming Service then the Server will exit.
 * Otherwise as long as at least one channel name has been registered
 * with the Naming Service the Event Server will continue.
 *
 * @param name The name of the event channel.
 * @param numberOfChannels The number of event channels specified.
 * @param bufferSize The size of the internal channel buffer, default 1000.
 * @param rebindFlag Determines if rebind command-line switch is specified.
 */
public void addChannelName(String name, int numberOfChannels,
    int bufferSize, boolean rebindFlag)
{
    NameComponent[] NC = new NameComponent[1];
    NameComponent[] channel_id = new NameComponent[2];
    boolean failStatus = false;

    NC[0] = new NameComponent(name, name);
    channel_id[0] = new NameComponent(name, name);
    channel_id[1] = new NameComponent(nameEC, nameEC);

    // create the event channel object.
    EventChannelImpl eventChannel = new EventChannelImpl(orb, this, bufferSize);

    try
    {

        channelContext = rootContext.bind_new_context(NC);

        // Register the Event Channel object reference with the
        // Naming Service.
        rootContext.bind(channel_id, eventChannel);

        // add the event channel object to the array.
        addEventChannel(eventChannel);
    }
    catch(InvalidName se)
    {
        System.out.println(NSError + "InvalidName : " + se.toString());
        System.out.println("The Name " + name + nameError);

        // keep count of the number of failed bind new contexts
        failChannelNumber++;
        // set the fail status flag
        failStatus = true;
    }
    catch(CannotProceed se)
    {
        System.out.println(NSError + "CannotProceed : " + se.toString());
        System.out.println("The Name " + name + nameError);
        failChannelNumber++;
        // set the fail status flag
        failStatus = true;
    }
    catch(NotFound se)
    {
        System.out.println(NSError + "NotFound : " + se.toString());
        System.out.println("The Name " + name + nameError);
        failChannelNumber++;
        // set the fail status flag
        failStatus = true;
    }
    catch(AlreadyBound se)
    {
        if (rebindFlag)
        { // the rebind flag is set so try to rebind the name.
            rebindChannelName(name, bufferSize);
        }
        else
        {

```

```

        System.out.println(NSError + "AlreadyBound : " + se.toString());
        System.out.println("The Name " + name + nameError);
        failChannelNumber++;
        // set the fail status flag
        failStatus = true;
    }
}
catch(org.omg.CORBA.SystemException se)
{
    System.out.println("Unexpected exception : " + se.toString());
    failChannelNumber++;
    // set the fail status flag
    failStatus = true;
}

if (failStatus)
{ // an exception has been thrown check if we need to
  // exit the server. We only exit if none of the channels
  // specified at the command-line have been entered into
  // the Naming Context tree.
  if (failChannelNumber == numberOfChannels)
  { // unable to create any contexts so exit server
    System.out.println("EventServer exiting...");
    System.exit(1);
  }
}

} // End of addChannelName

/**
 *
 * Adds the Channel Name that has been specified by Channel
 * Manager method createChannel.
 *
 * @param name The name of the event channel.
 *
 */
public boolean addChannelName(String name)
{
    NameComponent[] NC = new NameComponent[1];
    NameComponent[] channel_id = new NameComponent[2];
    boolean failStatus = false;

    NC[0] = new NameComponent(name, name);
    channel_id[0] = new NameComponent(name, name);
    channel_id[1] = new NameComponent(nameEC, nameEC);

    // create the event channel object, with a default buffer size, 1000.
    EventChannelImpl eventChannel = new EventChannelImpl(orb, this, 0);
    try
    {

        channelContext = rootContext.bind_new_context(NC);

        // Register the Event Channel object reference with the
        // Naming Service.
        rootContext.bind(channel_id, eventChannel);

        // add the event channel object to the array.
        addEventChannel(eventChannel);
    }
    catch(InvalidName se)
    {
        failStatus = true;
    }
    catch(CannotProceed se)
    {
        failStatus = true;
    }
    catch(NotFound se)
    {
        failStatus = true;
    }
    catch(AlreadyBound se)
    {
        failStatus = true;
    }
    catch(org.omg.CORBA.SystemException se)
    {
        failStatus = true;
    }
}

if (failStatus)
{ // an exception has been thrown so create channel
  // must now throw a duplicateChannel exception.
  return true;
}

```

```

    }
    else
    {
        return false;
    }
} // End of addChannelName

/**
 *
 * Removes the Channel Name from the Naming Service.
 * @param name The event channel context name to be removed.
 * @param channelRef The event channel to be removed.
 */
public void removeChannelName(String name, EventChannel channelRef)
{
    NameComponent[] NC = new NameComponent[1];
    NameComponent[] channel_id = new NameComponent[2];

    NC[0] = new NameComponent(name, name);
    channel_id[0] = new NameComponent(name, name);
    channel_id[1] = new NameComponent(nameEC, nameEC);

    try
    {
        rootContext.unbind(NC);
        channelRef.destroy();
    }
    catch(InvalidName se)
    {
        System.out.println(NSError + "InvalidName : " + se.toString());
        System.out.println("The Name " + name + nameError);
    }
    catch(CannotProceed se)
    {
        System.out.println(NSError + "CannotProceed : " + se.toString());
        System.out.println("The Name " + name + nameError);
    }
    catch(NotFound se)
    {
        System.out.println(NSError + "NotFound : " + se.toString());
        System.out.println("The Name " + name + nameError);
    }
    catch(org.omg.CORBA.SystemException se)
    {
        System.out.println("Unexpected exception : " + se.toString());
    }
} // End of removeChannelName

/**
 *
 * This method finds all the channel names that are registered
 * with the naming service. It gets all the context names that
 * are below the root context and then determines if they are
 * of type Event Channel. If they are, there added to a vector
 * array which is returned to the calling method.
 *
 * @return Vector An array of event channel names in the Naming Service.
 */
public Vector listAllChannelNames()
{
    Vector nameArray = new Vector();
    BindingListHolder blist = new BindingListHolder();
    BindingIteratorHolder biterHolder = new BindingIteratorHolder();
    BindingHolder binding = new BindingHolder();

    // get all the name context that are one level below the root.
    rootContext.list(1000,blist,biterHolder);

    // loop around all the name context's below the root context
    // and determine if it is of type EventChannel. If so add to
    // the vector array.
    for (int i=0; i < blist.value.length; i++)
    {
        String name = new String(blist.value[i].binding_name[0].id);
        EventChannel channel = resolveChannel.resolveChannelName(name, false);
        if (channel != null)
        {
            nameArray.addElement(name);
        }
    }
}

```



```

    return nameArray;
} // End of listAllChannelNames

/**
 *
 * This method finds all the channel names that are registered
 * with the naming service. It gets all the context names that
 * are below the root context and then determines the Event
 * channel object reference for each one. If the object matches
 * the object reference passed in then it's context name is
 * returned.
 *
 * @return The context name of the Event Channel object reference.
 */
public String findContextName(EventChannel channelRef)
{
    BindingListHolder blist = new BindingListHolder();
    BindingIteratorHolder biterHolder = new BindingIteratorHolder();
    BindingHolder binding = new BindingHolder();
    String ContextName = new String();

    // get all the name context that are one level below the root.
    rootContext.list(1000,blist,biterHolder);

    // loop around all the name context's below the root context
    // and determine if it is of type EventChannel. If so add to
    // the vector array.
    for (int i=0; i < blist.value.length; i++)
    {
        String name = new String(blist.value[i].binding_name[0].id);
        EventChannel channel = resolveChannel.resolveChannelName(name, false);
        if (channelRef.equals(channel))
        {
            ContextName = name;
            break;
        }
    }

    return ContextName;
} // End of findContextName

/**
 *
 * Gets the event channel implementation object reference that
 * is at the position specified in the vector array.
 *
 * @param index The index into the event channel vector array.
 * @return EventChannelImpl the event channel object in the array.
 */
public EventChannelImpl getEventChannel(int index)
{
    return ((EventChannelImpl)eventChannelArray.elementAt(index));
} // End of getEventChannel

/**
 *
 * This method returns the array vector of the Event Channel object
 * references.
 *
 * @return eventChannelArray Array of event channel object references.
 */
public final Vector getEventChannelArray()
{
    return eventChannelArray;
} // End of getEventChannelArray

/**
 *
 * Removes the event channel object reference from the vector array.
 *
 * @param channelRef The event channel object reference to be removed.
 */
public void removeEventChannel(EventChannel channelRef)
{
    int size = eventChannelArray.size();
    int index = 0;

```

```

boolean foundEventChannel = false;

// find the event channel object position in the array
for (int i=0; i<size; i++)
{
    if (eventChannelArray.elementAt(i).equals(channelRef))
    {
        index = i;
        foundEventChannel = true;
        break;
    }
}

if (foundEventChannel)
{
    eventChannelArray.removeElementAt(index);
}

} // End of removeEventChannel

/**
 *
 * Adds a Proxy push consumer to the push consumers list. The list
 * contains all the push consumers associated with the Event Channel.
 *
 * @param pConsumer The proxy push consumer to be added to the list.
 *
 */
public void addPushConsumer(ProxyPushConsumer pConsumer)
{
    push_consumers.addElement(pConsumer);
} // End of addPushConsumer

/**
 *
 * Removes a Proxy push consumer from the push consumers list. The list
 * contains all the push consumers associated with the Event Channel.
 *
 * @param pConsumer The proxy push consumer to be removed from the list.
 *
 */
public void removePushConsumer(ProxyPushConsumer pConsumer)
{
    int size = push_consumers.size();

    for (int i=0; i<size; i++)
    {
        if (push_consumers.elementAt(i).equals(pConsumer))
        {
            push_consumers.removeElementAt(i);
            break;
        }
    }
} // End of removePushConsumer

/**
 *
 * Adds a Proxy pull consumer to the pull consumers list. The list
 * contains all the pull consumers associated with the Event Channel.
 *
 * @param pConsumer The proxy pull consumer to be added to the list.
 *
 */
public void addPullConsumer(ProxyPullConsumer pConsumer)
{
    pull_consumers.addElement(pConsumer);
} // End of addPullConsumer

/**
 *
 * Removes a Proxy pull consumer from the pull consumers list. The list
 * contains all the pull consumers associated with the Event Channel.
 *
 * @param pConsumer The proxy pull consumer to be removed from the list.
 *
 */
public void removePullConsumer(ProxyPullConsumer pConsumer)
{
    int size = pull_consumers.size();

    for (int i=0; i<size; i++)
    {
        if (pull_consumers.elementAt(i).equals(pConsumer))

```

```

    {
        pull_consumers.removeElementAt(i);
        break;
    }
}

} // End of removePullConsumer

/**
 *
 * Returns the Vector of the references to the pull consumers.
 * @return The vector array containing the pull consumers references.
 */
public Vector getPullConsumers()
{
    return pull_consumers;
} // End of getPullConsumers

/**
 *
 * Adds a Proxy push supplier to the push suppliers list. The list
 * contains all the push suppliers associated with the Event Channel.
 * @param pSupplier The proxy push supplier to be added to the list.
 */
public void addPushSupplier(ProxyPushSupplier pSupplier)
{
    push_suppliers.addElement(pSupplier);
} // End of addPushSupplier

/**
 *
 * Removes a Proxy push supplier from the push suppliers list. The list
 * contains all the push supplier associated with the Event Channel.
 * @param pSupplier The proxy push supplier to be removed from the list.
 */
public void removePushSupplier(ProxyPushSupplier pSupplier)
{
    int size = push_suppliers.size();

    for (int i=0; i<size; i++)
    {
        if (push_suppliers.elementAt(i).equals(pSupplier))
        {
            push_suppliers.removeElementAt(i);
            break;
        }
    }
} // End of removePushSupplier

/**
 *
 * Returns the Vector of the references to the push suppliers.
 * @return The vector array containing the push suppliers references.
 */
public Vector getPushSuppliers()
{
    return push_suppliers;
} // End of getPushSuppliers

/**
 *
 * Adds a Proxy pull supplier to the pull suppliers list. The list
 * contains all the pull suppliers associated with the Event Channel.
 * @param pSupplier The proxy pull supplier to be added to the list.
 */
public void addPullSupplier(ProxyPullSupplier pSupplier)
{
    pull_suppliers.addElement(pSupplier);
} // End of addPullSupplier

```

```

/**
 *
 * Removes a Proxy pull supplier from the pull suppliers list. The list
 * contains all the pull supplier associated with the Event Channel.
 *
 * @param pSupplier The proxy pull supplier to be removed from the list.
 */
public void removePullSupplier(ProxyPullSupplier pSupplier)
{
    int size = pull_suppliers.size();

    for (int i=0; i<size; i++)
    {
        if (pull_suppliers.elementAt(i).equals(pSupplier))
        {
            pull_suppliers.removeElementAt(i);
            break;
        }
    }

} // End of removePullSupplier

/**
 *
 * Rebinds the Channel Name that has been already specified with
 * the Naming Service.
 *
 * @param name The name of the event channel.
 * @param bufferSize The size of the internal channel buffer, default 1000.
 */
private void rebindChannelName(String name, int bufferSize)
{
    NamingContext nameContext = null;
    org.omg.CORBA.Object object;

    NameComponent[] channel_id = new NameComponent[2];

    channel_id[0] = new NameComponent(name, name);
    channel_id[1] = new NameComponent(nameEC, nameEC);

    // create the event channel object.
    EventChannelImpl eventChannel = new EventChannelImpl(orb, this, bufferSize);

    try
    {
        rootContext.rebind(channel_id, eventChannel);

        // add the event channel object to the array.
        addEventChannel(eventChannel);
    }
    catch(InvalidName se)
    {
        System.out.println(NSError + "InvalidName : " + se.toString());
        System.out.println("Check the name, " + name + " ,unable to rebind name ");
    }
    catch(CannotProceed se)
    {
        System.out.println(NSError + "CannotProceed : " + se.toString());
        System.out.println("Check the name, " + name + " ,unable to rebind name ");
    }
    catch(NotFound se)
    {
        System.out.println(NSError + "NotFound : " + se.toString());
        System.out.println("Check the name, " + name + " ,unable to rebind name ");
    }
    catch(org.omg.CORBA.SystemException se)
    {
        System.out.println("Unexpected exception : " + se.toString());
        System.out.println("Check the name, " + name + " ,unable to rebind name ");
    }
} // End of rebindChannelName

/**
 *
 * Adds an event channel object reference to the event channel
 * vector array.
 *
 * @param eventChannel The event channel object reference to be added.
 */
private void addEventChannel(EventChannelImpl eventChannel)
{

```

```

        eventChannelArray.addElement(eventChannel);

    } // End of addEventChannel

} // End of class Channel

//-----
/*
CLASS ChannelManagerImpl extends _ChannelManagerImplBase
*/
//-----

package TCDEvents.EventsImpl;

import java.util.*;

import TCDEvents.CosEventChannelAdmin.*;
import TCDEvents.EventServerAdmin.*;

/**
 *
 * Every Event Server will have a Channel Manager object. The
 * Naming service is used to obtain a reference to this object.
 * The purpose of this class is to allow the user to create and
 * manipulate multiple Event Channels within the Event Server.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

public class ChannelManagerImpl extends _ChannelManagerImplBase
{

    private Channel eventChannel;
    private ResolveNames resolveChannel;

    /**
     *
     * Class constructor, initialises the locate event channel object.
     *
     * @param loc EventChannel object, used for locating event channels.
     *
     */

    public ChannelManagerImpl(Channel loc)
    {
        eventChannel = loc;

        // create the resolveNames object to resolve the channel name.
        resolveChannel = new ResolveNames();

    } // End of ChannelManagerImpl

    /**
     *
     * Creates an Event Channel within the Server application, and
     * registers it's name passed in by 'channelName' with the
     * Naming Service.
     *
     * @param channelName The name of the event channel to be added to NS.
     * @exception duplicateChannel The event channel already registered with NS.
     * @return A reference to the event channel object that's been created.
     *
     */

    public EventChannel createChannel(String channelName)
        throws duplicateChannel
    {
        if (eventChannel.addChannelName(channelName))
        {
            throw new duplicateChannel();
        }

        return (resolveChannel.resolveChannelName(channelName, true));

    } // End of createChannel

    /**
     *
     * Finds the Event Channel object reference that is associated
     * with the name 'channelName' that's passed in.
     *
     * @param channelName The name of the event channel to be found.
     * @exception noSuchChannel The event channel does not exist.
     *
     */

```

```

* @return A reference to the event channel object that's been found.
*
*/

public EventChannel findChannel(String channelName)
    throws noSuchChannel
{

    EventChannel channelRef = resolveChannel.resolveChannelName(channelName, false);
    if (channelRef == null)
    {
        throw new noSuchChannel();
    }

    return channelRef;

} // End of findChannel

/**
 *
 * Destroys the Event Channel and removes the Channel object
 * reference from the Naming Service.
 *
 * @param channelName The name of the event channel to be destroyed.
 * @exception noSuchChannel The event channel does not exist.
 *
 */

public void destroyChannel(String channelName)
    throws noSuchChannel
{

    // get the reference to the event channel to be destroyed.
    EventChannel channelRef = resolveChannel.resolveChannelName(channelName, false);

    if (channelName == null)
    {
        throw new noSuchChannel();
    }

    // Remove the event channel object reference from the array
    eventChannel.removeEventChannel(channelRef);

    eventChannel.removeChannelName(channelName, channelRef);

} // End of destroyChannel

/**
 *
 * Returns a list of all the Event Channels associated with the
 * Server application.
 *
 * @return An array that contains all the names of the event channels.
 *
 */

public String [] listAllChannels()
{

    Vector nameArray = eventChannel.listAllChannelNames();
    String [] channelNames = new String[nameArray.size()];

    for (int i=0; i<nameArray.size(); i++)
    {
        channelNames[i] = new String(nameArray.elementAt(i).toString());
    }

    return channelNames;

} // End of listAllChannels

/**
 *
 * Finds the Channel that has the object reference 'channelRef',
 * and returns its associated name.
 *
 * @param channelRef The event channel reference to be found.
 * @exception noSuchChannel The event channel does not exist.
 * @return The name of the event channel.
 *
 */

public String findChannelName(EventChannel channelRef)
    throws noSuchChannel
{

```

```

    String name = eventChannel.findContextName(channelRef);

    if (name.length() == 0)
    {
        throw new noSuchChannel();
    }

    return name;
} // End of findChannelName

} // End of class ChannelManagerImpl

//-----
/*
CLASS ConsumerAdminImpl extends _ConsumerAdminImplBase
*/
//-----

package TCDEvents.EventsImpl;

import TCDEvents.CosEventChannelAdmin.*;

/**
 *
 * The ConsumerAdminImpl interface defines the first step for
 * connecting consumers to the Event Channel. Clients use this
 * class to obtain proxy suppliers.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class ConsumerAdminImpl extends _ConsumerAdminImplBase
{
    private EventChannelImpl eventChannel;
    private Channel channel;

    /**
     *
     * Initialises the event channel implementation object and the channel
     * object.
     *
     * @param eventChannel The object reference to the channel implementation.
     * @param channel The object reference to the channel class.
     */

    public ConsumerAdminImpl(EventChannelImpl eventChannel, Channel channel)
    {
        this.eventChannel = eventChannel;
        this.channel = channel;
    } // End of ConsumerAdminImpl

    /**
     *
     * This method returns a ProxyPushSupplier object which is then used
     * to connect a Push-style consumer to the Event Channel.
     *
     * @return A handle to a push supplier object, used by push consumers.
     */

    public ProxyPushSupplier obtain_push_supplier()
    {
        ProxyPushSupplier pushSupplier = new ProxyPushSupplierImpl(eventChannel,channel);

        // add to the push supplier list.
        channel.addPushSupplier(pushSupplier);

        // start the Thread Pool.
        ThreadPool pool = new ThreadPool(eventChannel, channel, 12);
        pool.startPoolThread();

        return pushSupplier;
    } // End of obtain_push_supplier

    /**
     *
     * This method returns a ProxyPullSupplier object which is then used
     * to connect a Pull-style consumer to the Event Channel.
     */

```

```

* @return A handle to a pull supplier object, used by pull consumers.
*
*/

public ProxyPullSupplier obtain_pull_supplier()
{
    ProxyPullSupplier pullSupplier = new ProxyPullSupplierImpl(eventChannel, channel);

    // add to the pull supplier list.
    channel.addPullSupplier(pullSupplier);

    return pullSupplier;

} // End of obtain_pull_supplier

} // End of class ConsumerAdminImpl

//-----
/*
CLASS ConsumerThread extends Thread
*/
//-----

package TCDEvents.EventsImpl;

import java.util.*;
import org.omg.CORBA.Any;
import TCDEvents.CosEventChannelAdmin.*;

/**
 * This is the consumer thread that is part of the Thread
 * Pool. The Threads are all associated to the same Thread group.
 * The main purpose of the threads in the pool is to take event
 * data from the queue and deliver to all the connected consumers.
 *
 * @author Paul Stephens
 * @version 1.0
 */

class ConsumerThread extends Thread
{

    private SyncWaitBoundedQueue queue;
    private Channel channel;

    /**
     *
     * Creates the Thread and associates it with the group.
     *
     * @param queue The bounded queue object.
     * @param channel The object reference to the channel class.
     * @param group The Thread group object that the thread belongs too.
     */

    public ConsumerThread(SyncWaitBoundedQueue queue, Channel channel,
        ThreadGroup group)
    {
        super(group, "ConsumerThread");
        this.queue = queue;
        this.channel = channel;
    } // End of ConsumerThread

    /**
     *
     * The thread run method, takes items from the queue and delivers
     * them to all the consumers that are connected to the Event Channel.
     */

    public void run()
    {
        Any eventData;

        while(true)
        {
            if (queue.eventsPending() > 0)
            {
                Vector pushSuppliers = channel.getPushSuppliers();

                // get the event from the queue only if we have a
                // place to send it.
                if (pushSuppliers.size() > 0)

```



```

    {
        eventData = (Any)queue.takeEventItem();

        for(int i=0; i < pushSuppliers.size(); i++)
        {
            try
            {
                ((ProxyPushSupplierImpl)(pushSuppliers.elementAt(i))).receive(eventData);
            }
            catch(org.omg.CORBA.COMM_FAILURE ex)
            {
                // remove the push supplier from the list.
                channel.removePushSupplier((ProxyPushSupplier)pushSuppliers.elementAt(i));
            }
        }
    }
    else
    {
        // there's no proxy supplier, wait a while.
        try
        {
            Thread.sleep(1000*2);
        }
        catch(InterruptedException ex) {}
    }
}
else
{ // there's no events in the queue, wait a while.
    try
    {
        Thread.sleep(1000*2);
    }
    catch(InterruptedException ex) {}
}
try
{
    Thread.sleep(5);
}
catch(InterruptedException ex) {}
}

} // End of run

} // End of class ConsumerThread

//-----
/*
CLASS EventChannelImpl extends _EventChannelImplBase
*/
//-----

package TCDEvents.EventsImpl;

import org.omg.CORBA.ORB;
import TCDEvents.CosEventChannelAdmin.*;

/**
 *
 * This class provides operations for adding consumers, adding
 * suppliers and destroying the Event Channel. It also provides
 * means for determining the capacity of the Event Channel and
 * the setting of a new capacity size.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class EventChannelImpl extends _EventChannelImplBase
{
    private ORB orb;

    private SyncWaitBoundedQueue queue;

    private SupplierAdminImpl AdminSupplier;
    private ConsumerAdminImpl AdminConsumer;

    private Channel channel;

    /**
     *
     * Class constructor, creates the SupplierAdmin and ConsumerAdmin
     * object references.
     *
     * @param orb The Object Request Broker that its connected to.
     */
}

```

```

* @param channel The object reference to the channel class.
* @param bufferSize The size of the queue associated with the channel.
*
*/

public EventChannelImpl(ORB orb, Channel channel, int bufferSize)
{

    AdminSupplier = new SupplierAdminImpl(this, channel);
    AdminConsumer = new ConsumerAdminImpl(this, channel);

    this.orb = orb;
    this.channel = channel;

    if (bufferSize > 0)
    { // create the queue associated with the event channel
      // with the size specified at the command-line.
      queue = new SyncWaitBoundedQueue(bufferSize);
    }
    else
    {
      queue = new SyncWaitBoundedQueue();
    }
}

} // End of EventChannelImpl

/**
 *
 * Sets the capacity of the Event Channel.
 *
 * @param capacity Number of events allowed to be in the channel. Default 1000.
 *
 */

public void setChannelCapacity(int capacity)
{
    queue.setCapacity(capacity);
} // End of setChannelCapacity

/**
 *
 * Determines the capacity of the Event Channel.
 *
 * @return Number of events that are allowed to be in the channel at any one time.
 *
 */

public final int getChannelCapacity()
{
    return queue.getCapacity();
} // End of getChannelCapacity

/**
 *
 * Gets the queue object reference thats associated with every
 * event channel
 *
 * @return The queue object reference.
 *
 */

public SyncWaitBoundedQueue getQueue()
{
    return queue;
} // End of getQueue

/**
 *
 * The ConsumerAdmin interface allows consumers to be connected
 * to the Event Channel.
 *
 * @return A ConsumerAdmin object, allows consumers to retrieve handles to suppliers.
 *
 */

public ConsumerAdmin for_consumers()
{
    return ((ConsumerAdmin)AdminConsumer);
} // End of for_consumers

/**
 *
 * The SupplierAdmin interface allows suppliers to be connected

```

```

* to the Event Channel.
*
* @return A SupplierAdmin object, allows suppliers to retrieve handles to consumers.
*
*/

public SupplierAdmin for_suppliers()
{
    return ((SupplierAdmin)AdminSupplier);
} // End of for_suppliers

/**
 *
 * Releases all resources associated with the Channel.
 *
 */

public void destroy()
{
    // disconnect the event channel from the ORB
    orb.disconnect(this);

    // remove the event channel object reference from the vector array.
    channel.removeEventChannel((EventChannel)this);
} // End of destroy
} // End of class EventChannelImpl

//-----
/*
CLASS EventChannelMgr extends LocateNamingService
*/
//-----

package TCDEvents.EventsImpl;

import org.omg.CORBA.SystemException;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

import TCDEvents.EventServerAdmin.*;

/**
 * This class registers the channel manager object
 * with the Naming Service. The channel manager object
 * allows the user to create an manipulate multiple channels
 * within the server application.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

public class EventChannelMgr extends LocateNamingService
{
    private NamingContext rootContext = null;
    private NamingContext channelContext = null;

    /**
     *
     * Class constructor, resolves the Naming Service and it's
     * root context. The channel manager name is placed directly
     * below the root.
     *
     */
    public EventChannelMgr()
    {
        // get the root context of the naming service
        rootContext = getRootContext();
    } // End of EventChannelMgr

    /**
     *
     * This adds the Channel Manager object refernece to the Naming Service.
     * The Channel Manager object allows the user to create and manipulate
     * multiple event channels.
     *
     * @param channelMgr The Channel Manager object refernece
     *
     */
    public void addChannelMgr(ChannelManagerImpl channelMgr)

```

```

{
    NameComponent[] NC = new NameComponent[1];
    NameComponent[] channelMgr_id = new NameComponent[2];

    String name = new String("ChannelManager");
    String MgrName = new String("manager");

    NC[0] = new NameComponent(name, name);
    channelMgr_id[0] = new NameComponent(name, name);
    channelMgr_id[1] = new NameComponent(MgrName, MgrName);

    try
    {
        // create name "ChannelManager-ChannelManager"
        channelContext = rootContext.bind_new_context(NC);

        // Register the Channel Manager object reference with the
        // Naming Service.
        rootContext.bind(channelMgr_id, channelMgr);
    }
    catch(InvalidName se)
    {
        System.out.println(NSError + "InvalidName : " + se.toString());
        System.out.println("The Name " + name + nameError);
        System.out.println("EventServer exiting...");
        System.exit(1);
    }
    catch(CannotProceed se)
    {
        System.out.println(NSError + "CannotProceed : " + se.toString());
        System.out.println("The Name " + name + nameError);
        System.out.println("EventServer exiting...");
        System.exit(1);
    }
    catch(NotFound se)
    {
        System.out.println(NSError + "NotFound : " + se.toString());
        System.out.println("The Name " + name + nameError);
        System.out.println("EventServer exiting...");
        System.exit(1);
    }
    catch(AlreadyBound se)
    {
        rebindChannelMgr(name, MgrName, channelMgr);
    }
    catch(org.omg.CORBA.SystemException se)
    {
        System.out.println("Unexpected exception : " + se.toString());
        System.out.println("EventServer exiting...");
        System.exit(1);
    }
} // End of addChannelMgr

/**
 *
 * Rebinds the Channel Manager that has been already specified with
 * the Naming Service.
 *
 * @param name The context name of the Channel Manager.
 * @param MgrName The object name for the Channel Manager.
 * @param channelMgr The object reference for the Channel Manager object.
 */
private void rebindChannelMgr(String name, String MgrName,
    ChannelManagerImpl channelMgr)
{
    NameComponent[] mgr_id = new NameComponent[2];
    mgr_id[0] = new NameComponent(name, name);
    mgr_id[1] = new NameComponent(MgrName, MgrName);

    try
    {
        rootContext.rebind(mgr_id, channelMgr);
    }
    catch(InvalidName se)
    {
        System.out.println(NSError + "InvalidName : " + se.toString());
        System.out.println("Check the name, " + name + " ,unable to rebind name ");
        System.out.println("EventServer exiting...");
        System.exit(1);
    }
    catch(CannotProceed se)
    {
        System.out.println(NSError + "CannotProceed : " + se.toString());

```

```

        System.out.println("Check the name, " + name + " ,unable to rebind name ");
        System.out.println("EventServer exiting...");
        System.exit(1);
    }
    catch(NotFound se)
    {
        System.out.println(NSError + "NotFound : " + se.toString());
        System.out.println("Check the name, " + name + " ,unable to rebind name ");
        System.out.println("EventServer exiting...");
        System.exit(1);
    }
    catch(org.omg.CORBA.SystemException se)
    {
        System.out.println("Unexpected exception : " + se.toString());
        System.out.println("Check the name, " + name + " ,unable to rebind name ");
        System.out.println("EventServer exiting...");
        System.exit(1);
    }
} // End of rebindChannelMgr

} // End of class EventChannelMgr

//-----
/*
CLASS EventServer
*/
//-----

package TCDEvents.EventsImpl;

import java.util.*;

import org.omg.CORBA.ORB;
import IE.Iona.OrbixWeb._CORBA ;
import org.omg.CORBA.SystemException ;

/**
 *
 * This is the main entry point into the Event Server application.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class EventServer
{
    private Argument cmd_args;
    private Channel eventChannel;
    private EventChannelMgr chanMgr;
    private ORB orb;

    /**
     *
     * The main entry point into the Event Server application.
     *
     * @param args A string array that contains the command-line arguments.
     */

    public static void main(String args[])
    {
        EventServer es = new EventServer(args);
    } // End of main

    /**
     *
     * Creates the class Argument that handles the parsing
     * of the command-line arguments.
     *
     * @param args A string array that contains the command-line arguments.
     *
     * @see TCDEvents.EventsImpl.Argument
     * @see TCDEvents.EventsImpl.RegisterChannel
     * @see TCDEvents.EventsImpl.Channel
     */

    public EventServer(String args[])
    {

```

```

String ServerName = new String("ES");

// initialise the ORB
orb = ORB.init(args,null);

// create the Argument class. This handles all the
// parsing of the command-line arguments for the Event Server.
cmd_args = new Argument(args);

// loop through the command line arguments to determine
// the options that have been set.
for (int i =0; i < args.length; i++)
{
    // the first argument has to be the channel name.
    // there could be many channel names specified,
    // important to get them all.

    // check that we have a channel ID then assign it to the
    // array of names.

    cmd_args.determine_channel(i);

    // check if the rebind switch is specified.
    cmd_args.determine_rebind(i);

    // check is the server name switch is specified.
    if ( cmd_args.determine_serverName(i) )
    { // we have handled the server name correctly
        // now increment 'i' to skip the name argument.
        i++;
    }

    // check is the buffer switch is specified.
    if ( cmd_args.determine_buffer(i) )
    { // we have handled the buffer switch correctly
        // now increment 'i' to skip the size argument
        i++;
    }

    // check is the timeout switch is specified.
    if ( cmd_args.determine_timeout(i) )
    { // we have handled the timeout switch correctly
        // now increment 'i' to skip the size argument
        i++;
    }
}

// register the channel names with the Naming Service
eventChannel = new Channel(orb);
chanMgr = new EventChannelMgr();

// create the Channel Manager object, there is one per server
// application. Then add the name "ChannelManager" to the
// Naming Service.

ChannelManagerImpl channelMgr = new ChannelManagerImpl(eventChannel);
chanMgr.addChannelMgr(channelMgr);

String [] names = cmd_args.channel_names();
int numofChannels = cmd_args.number_of_channels();
int bufferSize = cmd_args.buffer_size();
boolean rebindFlag = cmd_args.rebind();

for (int i=0; i<numofChannels; i++)
{
    eventChannel.addChannelName(names[i], numofChannels, bufferSize, rebindFlag);
}
Vector eventChannelArray = eventChannel.getEventChannelArray();
int size = eventChannelArray.size();

for(int i=0; i < size; i++)
{
    orb.connect(eventChannel.getEventChannel(i));
}

if (cmd_args.server_name() != null)
{
    ServerName = cmd_args.server_name();
}

if (cmd_args.timeout_value() == 0)
{ // no timeout specified so use infinite timeout
    _CORBA.Orbix.impl_is_ready(ServerName, _CORBA.IT_INFINITE_TIMEOUT);
}
else

```

```

    {
        // convert the timeout into minutes.
        int timeout = (1000*60) * cmd_args.timeout_value();
        _CORBA.Orbix.impl_is_ready(ServerName, timeout);
    }

    System.out.println("Shutting down server...");

    for(int i=0; i<size; i++)
    {
        orb.disconnect(eventChannel.getEventChannel(i));
    }
    System.out.println ("Server exiting....");

} // End of EventServer

} // End of class EventServer

//-----
/*
CLASS LocateNamingService
*/
//-----

package TCDEvents.EventsImpl;

import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

/**
 *
 * This class resolves the Naming Service and it's root context.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

public class LocateNamingService
{
    private NamingContext rootContext = null;
    private org.omg.CORBA.Object initNCRRef;

    static final String NSErr = "Naming Service Exception - ";
    static final String nameError = " will not be added to the Naming Service";

    /**
     *
     * Class constructor, resolves the Naming Service and it's
     * root context.
     *
     */
    public LocateNamingService()
    {
        // Using default Naming Service (NS) host and port.
        // Default NS server name is 'NS'
        // Can be configured depending on the ORB that is being used.

        try
        {
            // obtain the Naming Service reference
            initNCRRef = ORB.init().resolve_initial_references("NameService");

            // resolve the reference to the Root context. All channel
            // names will be located one level below the Root.
            rootContext = NamingContextHelper.narrow(initNCRRef);
        }
        catch(org.omg.CORBA.ORBPackage.InvalidName in)
        {
            System.out.println("Exception during narrow of initial reference : " +
in.toString() );
            System.exit(1);
        }
        catch (SystemException ex)
        {
            System.out.println("Exception during connection to NamingService : " +
ex.toString() );
            System.exit(1);
        }
    }

} // End of LocateNamingService

```

```

/**
 *
 * Returns the root context of the naming service.
 *
 * @return The root context of the naming service.
 *
 */
public final NamingContext getRootContext()
{
    return rootContext;
} // End of getRootContext

} // End of class LocateNamingService

//-----
/**
CLASS ProxyPullConsumerImpl extends _ProxyPullConsumerImplBase
implements Runnable
*/
//-----

package TCDEvents.EventsImpl;

import java.util.*;

import TCDEvents.CosEventChannelAdmin.*;
import TCDEvents.CosEventComm.*;

import org.omg.CORBA.Any;
import org.omg.CORBA.BooleanHolder;

/**
 *
 * The ProxyPullConsumerImpl class defines the second step for
 * connecting pull suppliers to the Event Channel.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

public class ProxyPullConsumerImpl extends _ProxyPullConsumerImplBase
    implements Runnable
{
    private EventChannelImpl eventChannel;
    private Channel channel;

    private PullSupplier pull_supplier;
    private boolean connected = false;
    private final int sleepTime = 100;

    /**
     *
     * Initialises the event channel implementation and channel objects.
     *
     * @param eventChannel The object reference to the channel implementation.
     * @param channel The object reference to the channel class.
     *
     */

    public ProxyPullConsumerImpl(EventChannelImpl eventChannel, Channel channel)
    {
        this.eventChannel = eventChannel;
        this.channel = channel;
    } // End of ProxyPullConsumerImpl

    /**
     *
     * This method connects a pull supplier to the Event Channel.
     *
     * @param pull_supplier The pull supplier to connect to the Event Channel.
     * @exception AlreadyConnected If the pull supplier is already connected.
     *
     */

    public void connect_pull_supplier(PullSupplier pull_supplier)
        throws AlreadyConnected
    {
        if (pull_supplier == null)
        {
            throw new org.omg.CORBA.BAD_PARAM();
        }
    }
}

```



```

    }

    if (connected)
    {
        throw new AlreadyConnected();
    }

    this.pull_supplier = pull_supplier;

    connected = true;

    // start the thread running
    Thread tr = new Thread(this);
    tr.start();

} // End of connect_pull_supplier

/**
 *
 * Disconnects a pull consumer from the Event Channel.
 *
 */
public void disconnect_pull_consumer()
{
    if (connected)
    {
        connected = false;

        // remove the proxy pull consumer from the list
        channel.removePullConsumer((ProxyPullConsumer)this);
    }
} // End of disconnect_pull_consumer

/**
 *
 * Polls for an event and adds to the queue.
 *
 */
public void receive()
{
    Any eventData;
    BooleanHolder hasEvent = new BooleanHolder();

    try
    {
        eventData = pull_supplier.try_pull(hasEvent);
    }
    catch(org.omg.CORBA.COMM_FAILURE ex)
    {
        connected = false;
        return;
    }
    catch(Disconnected ex)
    {
        connected = false;
        return;
    }
}

// add event to the queue.
if (hasEvent.value == true)
{
    SyncWaitBoundedQueue queue = eventChannel.getQueue();
    queue.putEventItem(eventData);
}

} // End of receive

/**
 *
 * Polls for events.
 *
 */
public void run()
{
    while(connected)
    {
        try
        {
            Thread.sleep(sleepTime);
        }
        catch(InterruptedException ex) {}
    }
}

```

```

        receive();

        Thread.yield();
    }

} // End of run

} // End of class ProxyPullConsumerImpl

//-----
/*
CLASS ProxyPullSupplierImpl extends _ProxyPullSupplierImplBase
*/
//-----

package TCDEvents.EventsImpl;

import org.omg.CORBA.*;

import TCDEvents.CosEventChannelAdmin.*;
import TCDEvents.CosEventComm.*;

/**
 *
 * The ProxyPullSupplierImpl class defines the second step for
 * connecting pull consumers to the Event Channel.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

public class ProxyPullSupplierImpl extends _ProxyPullSupplierImplBase
{

    private EventChannelImpl eventChannel;
    private Channel channel;

    private PullConsumer pull_consumer;

    private boolean connected = false;

    /**
     *
     * Initialises the event channel implementation and channel objects.
     *
     * @param eventChannel The object reference to the channel implementation.
     * @param channel The object reference to the channel class.
     *
     */

    public ProxyPullSupplierImpl(EventChannelImpl eventChannel, Channel channel)
    {
        this.eventChannel = eventChannel;
        this.channel = channel;
    } // End of ProxyPullSupplierImpl

    /**
     *
     * If the Event Channel does not have any event data then it
     * invokes the method on every supplier that is connected. This
     * method will block until the supplier has generated an event,
     * causing the consumer to block.
     *
     * @return Any The event data.
     * @exception Disconnected If the supplier has been disconnected.
     *
     */

    public Any pull() throws Disconnected
    {
        Any eventData = null;
        BooleanHolder hasEvent = new BooleanHolder(false);

        while(hasEvent.value == false)
        {
            Thread.yield();
            eventData = try_pull(hasEvent);
            try
            {
                Thread.sleep(100);
            }
            catch(InterruptedException ex){}
        }
    }
}

```

```

    return eventData;

} // End of pull

/**
 *
 * This is similar to the pull method except that it is non-blocking.
 * If the supplier has event data it sets the hasEvent parameter
 * to true otherwise the parameter is set to false.
 *
 * @param hasEvent Determines if supplier has an event.
 * @return Any The event data.
 * @exception Disconnected If the supplier has been disconnected.
 */

public Any try_pull(BooleanHolder hasEvent) throws Disconnected
{
    // pull for the event

    if (connected)
    {
        synchronized(this)
        {
            SyncWaitBoundedQueue queue = eventChannel.getQueue();

            // determine if there are pending events in the queue
            hasEvent.value = (queue.eventsPending() != 0);

            if (hasEvent.value == true)
            {
                // get the event from the queue
                Any eventData = (Any)queue.takeEventItem();

                return eventData;
            }
            else
            {
                return ORB.init().create_any();
            }
        }
    }
    else
    {
        throw new Disconnected();
    }
} // End of try_pull

/**
 *
 * This method connects a pull consumer to the Event Channel.
 *
 * @param pull_consumer The pull consumer to connect to the Event Channel.
 * @exception AlreadyConnected If the pull consumer is already connected.
 */

public void connect_pull_consumer(PullConsumer pull_consumer)
    throws AlreadyConnected
{
    if (connected)
    {
        throw new AlreadyConnected();
    }

    if (pull_consumer != null)
    {
        this.pull_consumer = pull_consumer;
    }

    connected = true;
} // End of connect_pull_consumer

/**
 *
 * Disconnects a pull supplier from the Event Channel.
 */

public void disconnect_pull_supplier()
{

```

```

if (connected)
{
    connected = false;

    // removes the proxy pull supplier from the list
    channel.removePullSupplier((ProxyPullSupplier)this);

    // If the pull consumer is null then the consumer maybe
    // disconnected from the event channel without being informed
    if ( pull_consumer != null)
    {
        pull_consumer.disconnect_pull_consumer();
    }
}

} // End of disconnect_pull_supplier

} // End of class ProxyPullSupplierImpl

//-----
/*
CLASS ProxyPushConsumerImpl extends _ProxyPushConsumerImplBase
*/
//-----

package TCDEvents.EventsImpl;

import org.omg.CORBA.*;

import TCDEvents.CosEventChannelAdmin.*;
import TCDEvents.CosEventComm.*;

/**
 *
 * The ProxyPushConsumer interface defines the second step for
 * connecting push suppliers to the Event Channel.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class ProxyPushConsumerImpl extends _ProxyPushConsumerImplBase
{
    private EventChannelImpl eventChannel;
    private Channel channel;
    private PushSupplier push_supplier;

    private boolean connected = false;

    /**
     *
     * Initialises the event channel implementation and channel objects.
     *
     * @param eventChannel The object reference to the channel implementation.
     * @param channel The object reference to the channel class.
     */

    public ProxyPushConsumerImpl(EventChannelImpl eventChannel, Channel channel)
    {
        this.eventChannel = eventChannel;
        this.channel = channel;
    }

} // End of ProxyPushConsumerImpl

/**
 *
 * This method is used to transmit event data. The supplier invokes
 * this method to transfer the event data into the Event Channel.
 * Then the Event Channel invokes this method to transfer the
 * event data to all connected consumers.
 *
 * @param data The data to push into the Event Channel.
 * @exception Disconnected If the consumer has been disconnected.
 */

public void push(Any data) throws Disconnected
{
    SyncWaitBoundedQueue queue = eventChannel.getQueue();

    // place the event data into the queue.

```

```

    queue.putEventItem(data);

} // End of push

/**
 *
 * This method connects a push supplier to the Event Channel, so it
 * is ready to start sending event data.
 *
 * @param push_supplier The push supplier to connect to the Event Channel.
 * @exception AlreadyConnected If the push supplier is already connected.
 *
 */

public void connect_push_supplier(PushSupplier push_supplier)
    throws AlreadyConnected
{

    if (connected)
    { // the supplier is already connected, throw exception.
        throw new AlreadyConnected();
    }
    else
    {
        if (push_supplier != null)
        {
            this.push_supplier = push_supplier;
        }
        connected = true;
    }

} // End of connect_push_supplier

/**
 *
 * Disconnects a push consumer from the Event Channel.
 *
 */

public void disconnect_push_consumer()
{

    if (connected)
    {
        connected = false;

        // remove the proxy push consumer from the list
        channel.removePushConsumer((ProxyPushConsumer)this);

        // cannot invoke the disconnect_push_supplier if push_supplier is null.
        // The supplier can be disconnected without being informed
        if (push_supplier != null)
        {
            push_supplier.disconnect_push_supplier();
        }
    }

} // End of disconnect_push_consumer

} // End of class ProxyPushConsumerImpl

//-----
/**
CLASS ProxyPushSupplierImpl extends _ProxyPushSupplierImplBase
*/
//-----

package TCDEvents.EventsImpl;

import org.omg.CORBA.Any;

import TCDEvents.CosEventChannelAdmin.*;
import TCDEvents.CosEventComm.*;

/**
 *
 * The ProxyPushSupplierImpl class defines the second step for
 * connecting push consumers to the Event Channel.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

public class ProxyPushSupplierImpl extends _ProxyPushSupplierImplBase
{

```

```

private EventChannelImpl eventChannel;
private Channel channel;

private PushConsumer push_consumer;

private boolean connected = false;

/**
 *
 * Initialises the event channel implementation and channel objects.
 *
 * @param eventChannel The object reference to the channel implementation.
 * @param channel The object reference to the channel class.
 *
 */

public ProxyPushSupplierImpl(EventChannelImpl eventChannel, Channel channel)
{
    this.eventChannel = eventChannel;
    this.channel = channel;
} // End of ProxyPushSupplierImpl

/**
 *
 * This method connects a push consumer to the Event Channel, so it
 * is ready to receive event data.
 *
 * @param push_consumer The push consumer to connect to the Event Channel.
 * @exception AlreadyConnected If the push consumer is already connected.
 *
 */

public void connect_push_consumer(PushConsumer push_consumer)
    throws AlreadyConnected
{
    if (push_consumer == null)
    {
        throw new org.omg.CORBA.BAD_PARAM();
    }

    if (connected)
    {
        throw new AlreadyConnected();
    }

    this.push_consumer = push_consumer;

    connected = true;
} // End of connect_push_consumer

/**
 *
 * Disconnects a push supplier from the Event Channel.
 *
 */

public void disconnect_push_supplier()
{
    if (connected)
    {
        connected = false;

        // remove the push supplier from the list
        channel.removePushSupplier((ProxyPushSupplier)this);
    }
} // End of disconnect_push_supplier

/**
 *
 * Delivers the event data to the push consumer.
 *
 * @param data The event data to be delivered.
 *
 */

public void receive(Any data)
{
    // send event to push consumer.
    if (push_consumer != null)
    {
        try

```

```

    {
        push_consumer.push(data);
    }
    catch(Disconnected ex)
    {
        connected = false;
    }
}

} // End of receive

} // End of class ProxyPushSupplierImpl

//-----
/*
CLASS ResolveNames extends LocateNamingService
*/
//-----

package TCDEvents.EventsImpl;

import org.omg.CORBA.SystemException;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

import TCDEvents.CosEventChannelAdmin.*;
import TCDEvents.EventServerAdmin.*;

/**
 * This class is used to resolve the Event Channel Names
 * and the Channel Manager names of the Naming Service.
 * It is used by both the Supplier and Consumer applications.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class ResolveNames extends LocateNamingService
{
    private NamingContext rootContext = null;

    /**
     *
     * Class constructor, resolves the Naming Service and it's
     * root context.
     */
    public ResolveNames()
    {
        // get the root context of the naming service
        rootContext = getRootContext();
    } // End of ResolveNames

    /**
     *
     * This method resolves the name of the channel name that
     * passed in. It locates the name in the Naming context
     * tree of the Naming Service and returns its object
     * reference.
     *
     * @param channelName The context name to be resolved.
     * @param disMsg Display any error message that may arise.
     * @return An object reference to the resolved channel name.
     */
    public EventChannel resolveChannelName(String channelName, boolean disMsg)
    {
        EventChannel channel = null;
        org.omg.CORBA.Object obj;
        boolean status = false;

        try
        {
            NameComponent[] name = new NameComponent[2];
            name[0] = new NameComponent(channelName, channelName);
            name[1] = new NameComponent("eventChannel", "eventChannel");
            obj = rootContext.resolve(name);
            channel = EventChannelHelper.narrow(obj);
        }
        catch(NotFound se)
        {
            if (disMsg)
            {

```

```

        System.out.println("Exception - NotFound : " + se.toString());
    }
    status = true;
}
catch(InvalidName se)
{
    if (disMsg)
    {
        System.out.println("Exception - InvalidName : " + se.toString());
    }
    status = true;
}
catch(CannotProceed se)
{
    if (disMsg)
    {
        System.out.println("Exception - CannotProceed : " + se.toString());
    }
    status = true;
}
catch (SystemException ex)
{
    if (disMsg)
    {
        System.out.println("System Exception : " + ex.toString() );
    }
    status = true;
}
if (status)
{
    return null;
}
else
{
    return channel;
}
} // End of resolveChannelName

/**
 *
 * This method resolves the name of the channel manager
 * It locates the channel manager name context in the
 * Naming context tree of the Naming Service and returns
 * its object reference.
 *
 * @return An object reference to the resolved channel manager.
 */
public ChannelManager resolveChannelMgr()
{
    ChannelManager channelMgr = null;
    org.omg.CORBA.Object obj;
    String ChannelMgr_name = new String("ChannelManager");
    String ChannelMgr_obj = new String("manager");
    boolean status = false;
    try
    {
        NameComponent[] name = new NameComponent[2];
        name[0] = new NameComponent(ChannelMgr_name, ChannelMgr_name);
        name[1] = new NameComponent(ChannelMgr_obj, ChannelMgr_obj);
        obj = rootContext.resolve(name);
        channelMgr = ChannelManagerHelper.narrow(obj);
    }
    catch(NotFound se)
    {
        System.out.println("Exception - NotFound : " + se.toString());
        status = true;
    }
    catch(InvalidName se)
    {
        System.out.println("Exception - InvalidName : " + se.toString());
        status = true;
    }
    catch(CannotProceed se)
    {
        System.out.println("Exception - CannotProceed : " + se.toString());
        status = true;
    }
    catch (SystemException ex)
    {
        System.out.println("System Exception : " + ex.toString() );
        status = true;
    }
    if (status)
    {

```



```

        return null;
    }
    else
    {
        return channelMgr;
    }

} // End of resolveChannelMgr

} // End of class ResolveNames

//-----
/*
CLASS SupplierAdminImpl extends _SupplierAdminImplBase
*/
//-----

package TCDEvents.EventsImpl;

import java.util.*;

import TCDEvents.CosEventChannelAdmin.*;

/**
 *
 * The SupplierAdminImpl class defines the first step for
 * connecting suppliers to the Event Channel. Clients use this
 * class to obtain handles to proxy consumers.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class SupplierAdminImpl extends _SupplierAdminImplBase
{

    private EventChannelImpl eventChannel;
    private Channel channel;

    /**
     *
     * Initialises the event channel implementation object and the channel
     * object.
     *
     * @param eventChannel The object reference to the channel implementation.
     * @param channel The object reference to the channel class.
     */

    public SupplierAdminImpl(EventChannelImpl eventChannel, Channel channel)
    {
        this.eventChannel = eventChannel;
        this.channel = channel;
    } // End of SupplierAdminImpl

    /**
     *
     * This method returns a ProxyPushConsumer object which is then
     * used to connect a Push-style supplier to the Event Channel.
     *
     * @return ProxyPushConsumer object reference.
     */

    public ProxyPushConsumer obtain_push_consumer()
    {
        ProxyPushConsumer pushConsumer = new ProxyPushConsumerImpl(eventChannel, channel);

        // add to the push consumer list.
        channel.addPushConsumer(pushConsumer);

        return pushConsumer;
    } // End of obtain_push_consumer

    /**
     *
     * This method returns a ProxyPullConsumer object which is then
     * used to connect a Pull-style supplier to the Event Channel.
     *
     * @return ProxyPullConsumer object reference.
     */
}

```

```

public ProxyPullConsumer obtain_pull_consumer()
{
    ProxyPullConsumer pullConsumer = new ProxyPullConsumerImpl(eventChannel, channel);

    // add to the pull consumer list.
    channel.addPullConsumer(pullConsumer);

    return pullConsumer;
} // End of obtain_pull_consumer

/**
 *
 * Pulls the event data from the supplier.
 *
 */
synchronized void pull()
{
    Vector pullConsumer = channel.getPullConsumers();

    for (int i=0; i < pullConsumer.size(); i++)
    {
        try
        {
            ((ProxyPullConsumerImpl)(pullConsumer.elementAt(i))).receive();
        }
        catch(org.omg.CORBA.COMM_FAILURE ex)
        {
            // remove the pull consumer from the list.
            channel.removePullConsumer((ProxyPullConsumer)pullConsumer.elementAt(i));
        }
    }
} // End of pull
} // End of class SupplierAdminImpl

//-----
/**
CLASS SyncWaitBoundedQueue extends BoundedQueue
*/
//-----

package TCDEvents.EventsImpl;

import java.util.*;

/**
 * This class is a thread safe implementation of the BoundedQueue
 * class. It allows event data to be added to the queue and
 * removed from the queue. Suppliers will add data to the queue
 * and consumers will remove data from the queue. The default
 * size of the queue is 1000.
 *
 *
 * @author Paul Stephens
 * @version 1.0
 *
 * @see BoundedQueue
 *
 */

public class SyncWaitBoundedQueue extends BoundedQueue
{
    /**
     *
     * Calls the constructor for the BoundedQueue object.
     * This will set the queue to the default size of 1000.
     *
     */

    public SyncWaitBoundedQueue()
    {
        super();
    } // End of SyncWaitBoundedQueue

    /**
     *
     * Calls the constructor for the BoundedQueue object.
     * This will set the queue to the size specified.
     *
     */

```

```
public SyncWaitBoundedQueue(int size)
{
    super(size);
} // End of SyncWaitBoundedQueue

/**
 *
 * Determines if the queue is empty.
 * It calls the isEmpty method of the BoundedQueue class.
 *
 * @return True if the queue is empty, otherwise false.
 * @see BoundedQueue
 */

public synchronized boolean isEmpty()
{
    return super.isEmpty();
} // End of isEmpty

/**
 *
 * Determines if the queue is full.
 * It calls the isFull method of the BoundedQueue class.
 *
 * @return True if the queue is full, otherwise false.
 * @see BoundedQueue
 */

public synchronized boolean isFull()
{
    return super.isFull();
} // End of isFull

/**
 *
 * The number of events in the queue that are pending delivery.
 *
 * @return The number of events that are pending delivery.
 * @see BoundedQueue
 */

public synchronized int eventsPending()
{
    return super.eventsPending();
} // End of eventsPending

/**
 *
 * This method gets the size of the queue.
 *
 * @return The size of the queue.
 * @see BoundedQueue
 */

public int getCapacity()
{
    return super.getCapacity();
} // End of getCapacity

/**
 *
 * This method sets the size of the queue.
 *
 * @param capacity The new size of the queue.
 * @see BoundedQueue
 */

public synchronized void setCapacity(int capacity)
{
    super.setCapacity(capacity);
} // End of setCapacity
```

```

/**
 *
 * Adds an event data item to the back of the queue.
 * Determines if there is room in the queue to add
 * another event data item. If not then the method
 * waits until it has been notified that an item has
 * been removed for the queue and delivered to the
 * connecting consumers. Once there is room for the event
 * data it gets added to the back of the queue.
 *
 * @param obj The event data item object.
 * @see BoundedQueue
 *
 */

public synchronized void putEventItem(Object obj)
{
    try
    {
        while (isFull())
        { // wait until item removed from queue
            wait();
        }
    }
    catch (InterruptedException e) {}
    super.putEventItem(obj);
    // notify waiting threads that there is
    // an item in the queue to be removed.
    notify();
} // End of putEventItem

/**
 *
 * Takes an event data item from the front of the queue.
 * Determines if the queue contains event data items
 * If it does not then the method waits until it an
 * item has been added, which notifies this method.
 * Once there is event items in the queue then the
 * event data gets removed from the front of the queue,
 * and will be delivered to the connecting consumers.
 *
 * @return The event data item that has been removed.
 * @see BoundedQueue
 *
 */

public synchronized Object takeEventItem()
{
    try
    {
        while (isEmpty())
        { // nothing in the queue to get so wait
            wait();
        }
    }
    catch (InterruptedException e) {}
    Object result = super.takeEventItem();
    notify();
    return result;
} // End of takeEventItem

} // End of class SyncWaitBoundedQueue

//-----
/**
CLASS ThreadPool
*/
//-----

package TCDEvents.EventsImpl;

/**
 *
 * This is the Thread Pool class, it creates the group that
 * the consumer threads are associated with. These threads
 * main purpose is to deliver event data to all connected
 * consumers. This class also starts the threads that are
 * members of the group.
 *
 * @author Paul Stephens
 * @version 1.0
 *
 */

```

```
public class ThreadPool
{
    private SyncWaitBoundedQueue queue;
    private EventChannelImpl eventChannel;
    private Channel channel;
    private ThreadGroup group = null;
    private ConsumerThread [] ct;
    private int numberOfThreads = 0;

    /**
     *
     * Creates the Thread group and an array of threads of the group.
     *
     * @param eventChannel The object reference to the channel implementation.
     * @param channel The object reference to the channel class.
     * @param numberOfThreads The number of threads in the group.
     */
    public ThreadPool(EventChannelImpl eventChannel, Channel channel,
        int numberOfThreads)
    {
        this.eventChannel = eventChannel;
        this.channel = channel;
        this.queue = eventChannel.getQueue();
        this.numberOfThreads = numberOfThreads;
        group = new ThreadGroup("Consumer Group");
        ct = new ConsumerThread[numberOfThreads];
    } // End of ThreadPool

    /**
     *
     * Creates and starts each thread in the Pool group.
     * For each thread in the pool group, it creates the thread
     * and starts it running.
     */
    public void startPoolThread()
    {
        for(int i=0; i<numberOfThreads; i++)
        {
            ct[i] = new ConsumerThread(queue, channel, group);
            ct[i].start();
        }
    } // End of startPoolThread

    /**
     *
     * Gets the group that the threads of the pool belong too.
     *
     * @return The group associated with the threads.
     */
    public final ThreadGroup getGroup()
    {
        return group;
    } // End of getGroup
} // End of class ThreadPool
```

GUI Tool Source Code

```
//-----
/*
CLASS FileCreateException extends Exception
*/
//-----

package TCDEvents.GUITools;

/**
 * Exception class for when the wizard GUI application
 * is unable to create the specified supplier/consumer
 * files. The most likely reason for this exception
 * to be thrown is only having read permission on the
 * files that have to be written too.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class FileCreateException extends Exception
{
    /**
     * Prints out the exception message to the screen.
     * This gets called whenever the consumer or supplier files
     * cause an error when getting created.
     *
     * @param fileName The name of the file that caused the problem.
     */
    public FileCreateException(String fileName)
    {
        System.err.println("Error : Unable to create file " + fileName);
        System.err.println("");
        System.err.println("The application will not be generated.");
        System.err.println("Check if file already exists, and its permissions.");
    }
    // End of FileCreateException
}
// End of class FileCreateException

//-----
/*
CLASS FileOutput
*/
//-----

package TCDEvents.GUITools;

import java.io.*;
import java.util.*;

/**
 * This is the class that handles the code generation
 * for the WizardGUI application. It will generate
 * either or both supplier/consumer applications
 * depending on the user selection. This class gets
 * called from WizardGUI.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class FileOutput
{
    private FileOutputStream consumerOut; // file output object for consumer
    private FileOutputStream supplierOut; // file output object for supplier

    private PrintStream pConsumer; // print stream object for consumer
    private PrintStream pSupplier; // print stream object for supplier

    private String channelName; // Name of the channel
    private int size = 0; // The size of the internal buffer

    /**
     * Creates either the Supplier or the consumer application files.
     * It connects the print streams to the file ready for writing.
     * Supplier application or Consumer application, it also
     * creates the file for locating the Event channel. The

```

```

* classes in this file will be used by both the supplier
* and consumer applications.
*
* @param fileName The file name for the supplier or consumer application.
* @param consumerFlag If true creates consumer file, otherwise supplier application is created.
* @param supplierPush True for push supplier, otherwise false.
* @param consumerPush True for push consumer, otherwise false.
* @param channelName The name of the channel to be resolved.
* @param bufferSize The size of the Event Channel internal buffer.
*
* @exception FileCreateException Error creating the consumer/supplier file.
*
* @see TCDEvents.GUITools.FileCreateException
*
*/

```

```

public FileOutput(String fileName, boolean consumerFlag,
    boolean supplierPush, boolean consumerPush,
    String channelName, String bufferSize)
    throws FileCreateException
{
    this.channelName = channelName;
    Integer size = new Integer(bufferSize);
    this.size = size.intValue();

    if (consumerFlag)
    {
        try
        {
            // Create a new file output stream for consumer
            consumerOut = new FileOutputStream(fileName);

            // Connect print stream to the output streams
            pConsumer = new PrintStream(consumerOut);
        }
        catch (Exception e)
        {
            System.err.println ("Error : Writing to file " + fileName);
            throw new FileCreateException(fileName);
        }

        // generate the code for the consumer application
        generateConsumer(consumerPush, fileName);
    }
    else
    {
        try
        {
            // Create a new file output stream for supplier
            supplierOut = new FileOutputStream(fileName);

            // Connect print stream to the output streams
            pSupplier = new PrintStream(supplierOut);
        }
        catch (Exception e)
        {
            System.err.println ("Error : Writing to file " + fileName);
            throw new FileCreateException(fileName);
        }

        // generate the code for the supplier application
        generateSupplier(supplierPush, fileName);
    }
} // End of FileOutput

```

```

/**
* Creates both supplier and consumer application files.
* It connects the print streams to it to both files for writing. It
* also creates the file for locating the Event channel.
*
* @param consumerName The file name for the consumer application.
* @param supplierName The file name for the supplier application.
* @param supplierPush True for push supplier, otherwise false.
* @param consumerPush True for push consumer, otherwise false.
* @param channelName The name of the channel to be resolved.
* @param bufferSize The size of the Event Channel internal buffer.
*
* @exception FileCreateException Error creating the consumer/supplier file.
*
* @see TCDEvents.GUITools.FileCreateException
*
*/

```

```

public FileOutput(String consumerName, String supplierName,

```

```

        boolean supplierPush, boolean consumerPush,
        String channelName, String bufferSize)
        throws FileCreateException
    {
        this.channelName = channelName;
        Integer size = new Integer(bufferSize);
        this.size = size.intValue();

        try
        {
            // Create a new file output streams for consumer
            consumerOut = new FileOutputStream(consumerName);

            // Connect print stream to the output stream
            pConsumer = new PrintStream( consumerOut );
        }
        catch (Exception e)
        {
            System.err.println ("Error : Writing to file " + consumerName);
            throw new FileCreateException(consumerName);
        }

        try
        {
            // Create a new file output streams for supplier
            supplierOut = new FileOutputStream(supplierName);

            // Connect print stream to the output stream
            pSupplier = new PrintStream( supplierOut );
        }
        catch (Exception e)
        {
            System.err.println ("Error : Writing to file " + supplierName);
            throw new FileCreateException(supplierName);
        }

        // generate the file and code for locating event channel
        generateConsumer(consumerPush, consumerName);
        generateSupplier(supplierPush, supplierName);
    } // End of FileOutput

/**
 * Generates the Java code for the consumer application.
 *
 * @param pushState True if a push consumer to be generated.
 * @param fileName The file name of the consumer application.
 */
private void generateConsumer(boolean pushState, String fileName)
{
    pConsumer.println("//");
    pConsumer.println("// Consumer Application - Java generated by the WizardGUI application");
    pConsumer.println("//");

    StringTokenizer tok = new StringTokenizer(fileName, ".");
    String className = tok.nextToken();

    if (pushState)
    {
        generatePushConsumer(className);
    }
    else
    {
        generatePullConsumer(className);
    }

    // close the connection
    pConsumer.close();
} // End of generateConsumer

/**
 * Generates the Java code for the supplier application.
 *
 * @param pushState True if a push supplier to be generated.
 * @param fileName The file name of the supplier application.
 */
private void generateSupplier(boolean pushState, String fileName)
{
    pSupplier.println("//");
    pSupplier.println("// Supplier Application - Java generated by the WizardGUI application");

```



```

pSupplier.println("//");

StringTokenizer tok = new StringTokenizer(fileName, ".");
String className = tok.nextToken();

if (pushState)
{
    generatePushSupplier(className);
}
else
{
    generatePullSupplier(className);
}

// close the connection
pSupplier.close();

} // End of generateSupplier

/**
 * Generates the Java code for the push consumer application.
 *
 * @param className The name of the class for the push consumer.
 *
 */
private void generatePushConsumer(String className)
{
    pConsumer.println("// Push Consumer");
    pConsumer.println("//");
    pConsumer.println("");
    pConsumer.println("import TCDEvents.EventsImpl.*;");
    pConsumer.println("import TCDEvents.CosEventChannelAdmin.*;");
    pConsumer.println("import TCDEvents.CosEventComm.*;");
    pConsumer.println("import org.omg.CORBA.Any;");
    pConsumer.println("");
    pConsumer.println("public class " + className + " extends _PushConsumerImplBase");
    pConsumer.println("{");
    pConsumer.println("    private ProxyPushSupplier pushSupplier;");
    pConsumer.println("");
    pConsumer.println("    static public void main(String[] args)");
    pConsumer.println("    {");
    pConsumer.println("");
    pConsumer.println("        " + className + " pushConsumer = null;");
    pConsumer.println("        ResolveNames resolveChannel;");
    pConsumer.println("        EventChannel channel;");
    pConsumer.println("        ConsumerAdmin consumer;");
    pConsumer.println("        ProxyPushSupplier pushSupplier;");
    pConsumer.println("");
    pConsumer.println("    //");
    pConsumer.println("    // Step 1. Get a ProxyPushSupplier object reference");
    pConsumer.println("    //");
    pConsumer.println("    // Obtain the event channel reference");
    pConsumer.println("    resolveChannel = new ResolveNames();");
    pConsumer.println("    String ec = channel = resolveChannel.resolveChannelName(\"");
    pConsumer.println("    ec = ec + channelName + \"\" + \"true\" + \");");
    pConsumer.println("    consumer =");
    pConsumer.println("    ");
    pConsumer.println("    if (channel == null)");
    pConsumer.println("    {");
    pConsumer.println("        System.out.println(\"Unable to resolve channel - exiting...\");");
    pConsumer.println("        System.exit(0);");
    pConsumer.println("    }");
    pConsumer.println("    if ((size != 1000) && (size >= 1))");
    pConsumer.println("    {");
    pConsumer.println("        ");
    pConsumer.println("        Setting the channel buffer size");
    pConsumer.println("        channel.setChannelCapacity(" + size + ");");
    pConsumer.println("    }");
    pConsumer.println("    ");
    pConsumer.println("    // Obtain a consumer administration object");
    pConsumer.println("    consumer = channel.for_consumers();");
    pConsumer.println("    ");
    pConsumer.println("    // Obtain a proxy push supplier");
    pConsumer.println("    pushSupplier = consumer.obtain_push_supplier();");
    pConsumer.println("    ");
    pConsumer.println("    pushConsumer = new " + className + "(pushSupplier);");
    pConsumer.println("    ");
    pConsumer.println("    //");
    pConsumer.println("    // Step 2. Connect to the Event Channel");
    pConsumer.println("    //");
    pConsumer.println("    ");
    pConsumer.println("    try");
    pConsumer.println("    {");
    pConsumer.println("        pushSupplier.connect_push_consumer((PushConsumer)pushConsumer);");
    pConsumer.println("    }");

```



```

pConsumer.println("\t\t\tSystem.out.println(\"Unable to resolve channel - exiting...\");");
pConsumer.println("\t\t\tSystem.exit(0);");
pConsumer.println("\t\t");
if ((size != 1000) && (size >= 1))
{
    pConsumer.println("");
    pConsumer.println("\t\t// Setting the channel buffer size");
    pConsumer.println("\t\tchannel.setChannelCapacity(" + size + ");");
}
pConsumer.println("");
pConsumer.println("\t\t// Obtain a consumer administration object");
pConsumer.println("\t\tconsumer = channel.for_consumers();");
pConsumer.println("");
pConsumer.println("\t\t// Obtain a proxy push supplier");
pConsumer.println("\t\tpullSupplier = consumer.obtain_pull_supplier();");
pConsumer.println("");
pConsumer.println("\t\tpullConsumer = new " + className + "(pullSupplier);");
pConsumer.println("");
pConsumer.println("\t\t/");
pConsumer.println("\t\t// Step 2. Connect to the Event Channel");
pConsumer.println("\t\t/");
pConsumer.println("");
pConsumer.println("\t\ttry");
pConsumer.println("\t\t{");
pConsumer.println("\t\tpullSupplier.connect_pull_consumer((PullConsumer)pullConsumer);");
pConsumer.println("\t\t}");
pConsumer.println("\t\tcatch(AlreadyConnected se)");
pConsumer.println("\t\t{");
pConsumer.println("\t\t\tSystem.out.println(\"AlreadyConnected - Pull consumer already connected.\");");
pConsumer.println("\t\t\tSystem.out.println(se.toString());");
pConsumer.println("\t\t\tSystem.exit(1);");
pConsumer.println("\t\t}");
pConsumer.println("");
pConsumer.println("\t\tSystem.out.println(\"Pull Consumer registered with the event channel.\");");
pConsumer.println("");
pConsumer.println("\t\t/");
pConsumer.println("\t\t// Step 3. Pull events from the Event Channel.");
pConsumer.println("\t\t/");
pConsumer.println("");
pConsumer.println("\t\tThread tr = new Thread(pullConsumer);");
pConsumer.println("\t\ttr.start();");
pConsumer.println("\t\t");
pConsumer.println("\t\t");
pConsumer.println("\t\tpublic " + className + "(ProxyPullSupplier pullSupplier)");
pConsumer.println("\t\t{");
pConsumer.println("\t\t\tthis.pullSupplier = pullSupplier;");
pConsumer.println("\t\t}");
pConsumer.println("\t\t");
pConsumer.println("\t\tpublic void disconnect_pull_consumer()");
pConsumer.println("\t\t{");
pConsumer.println("\t\t\tpullSupplier.disconnect_pull_supplier();");
pConsumer.println("\t\t\tpullSupplier = null;");
pConsumer.println("\t\t}");
pConsumer.println("\t\t");
pConsumer.println("\t\tpublic void run()");
pConsumer.println("\t\t{");
pConsumer.println("\t\t\twhile(pullSupplier != null)");
pConsumer.println("\t\t\t{");
pConsumer.println("\t\t\t\tAny eventData = null;");
pConsumer.println("\t\t\t\t");
pConsumer.println("\t\t\t\ttry");
pConsumer.println("\t\t\t\t{");
pConsumer.println("\t\t\t\t\teventData = pullSupplier.pull();");
pConsumer.println("\t\t\t\t\tString str = eventData.extract_string();");
pConsumer.println("\t\t\t\t\tSystem.out.println(str);");
pConsumer.println("\t\t\t\t\t");
pConsumer.println("\t\t\t\t\tcatch(Disconnected ex)");
pConsumer.println("\t\t\t\t\t{");
pConsumer.println("\t\t\t\t\t\tSystem.out.println(\"The pull supplier is disconnected.\");");
pConsumer.println("\t\t\t\t\t\treturn;");
pConsumer.println("\t\t\t\t\t\t");
pConsumer.println("\t\t\t\t\t\tcatch(COMM_FAILURE ex)");
pConsumer.println("\t\t\t\t\t\t{");
pConsumer.println("\t\t\t\t\t\t\tthrow new COMM_FAILURE();");
pConsumer.println("\t\t\t\t\t\t\t");
pConsumer.println("\t\t\t\t\t\t");
pConsumer.println("\t\t\t\t\t\tThread.yield();");
pConsumer.println("\t\t\t\t\t\t");
pConsumer.println("\t\t\t\t\t\ttry");
pConsumer.println("\t\t\t\t\t\t{");
pConsumer.println("\t\t\t\t\t\t\tThread.sleep(1000);");
pConsumer.println("\t\t\t\t\t\t\t");
pConsumer.println("\t\t\t\t\t\t\tcatch(InterruptedException ex) {}");
pConsumer.println("\t\t\t\t\t\t\t");
pConsumer.println("\t\t\t\t\t\t}");
pConsumer.println("\t\t\t\t\t}");
}
}

```



```

//-----
/*
CLASS WizardGUI extends Frame
*/
//-----

package TCDEvents.GUITools;

import java.awt.*;
import git.Box;

/**
 * This is the main class for the Wizard GUI
 * application. This application generates code for
 * locating the Event Channel, and connecting a
 * supplier or consumer to the channel. It steps the
 * user through each process, the user is not able to
 * proceed to the next stage until all information is
 * complete.
 *
 * @author Paul Stephens
 * @version 1.0
 */

public class WizardGUI extends Frame
{
    // objects for the card layout, the wizard has 5 different
    // screens labelled card_1 to card_5. These are placed on the
    // card stack.
    private Panel cardStack;
    private Panel card_1;
    private Panel card_2;
    private Panel card_3;
    private Panel card_4;
    private Panel card_5;
    private CardLayout cl;

    // standard check boxes for the supplier/consumer and there
    // communication model.
    private Checkbox consumerCB;
    private Checkbox supplierCB;
    private Checkbox consumerPushCB;
    private Checkbox consumerPullCB;
    private Checkbox supplierPushCB;
    private Checkbox supplierPullCB;

    // Text fields for the consumer, supplier, event channel and buffer.
    private TextField consumerTF;
    private TextField supplierTF;
    private TextField channelTF;
    private TextField bufferTF;

    // Text area to hold the summary information
    private TextArea summaryTA;

    // The standard labels for the supplier and consumer
    private Label consumerL1;
    private Label supplierL1;
    private Label consumerL2;
    private Label supplierL2;

    // The different buttons to navigate through the different screens
    // and to generate the required applications.
    private Button backB;
    private Button forwardB;
    private Button finishB;
    private Button cancelB;

    // File names for the supplier and consumer applications
    private String consumerFileName = new String();
    private String supplierFileName = new String();

    /**
     * The main enter point into the application, calls the
     * class constructor and sets the initial frame size.
     *
     * @param args A string array that contains the command-line arguments.
     */

    public static void main(String args[])
    {
        WizardGUI app = new WizardGUI("Supplier/Consumer Wizard Application");

        // Set the size of the initial frame and display it

```

```

app.reshape(100,100,400,180);
app.show();

} // End of main

/**
 * Initialises the frame, sets the frame to use the border layout,
 * then creates a card layout. This is to place the various screens
 * of the card stack to enable toggling between the different screens.
 *
 * @param title The title of the application displayed on the frame.
 *
 */

public WizardGUI(String title)
{

    super(title);
    // create a border layout for the main frame
    setLayout(new BorderLayout());

    // create a panel to hold the card stack. The card
    // stack will contain different panels for the
    // different screens
    cardStack = new Panel();

    // Create the card layout for the card stack and
    // add to the center.
    cl = new CardLayout();
    cardStack.setLayout(cl);
    add("Center", cardStack);

    // start creating all the screens and buttons.
    CreateScreen1();
    CreateScreen2();
    CreateScreen3();
    CreateScreen4();
    CreateScreen5();

    CreateButtons();

    backB.disable();
    finishB.disable();

} // End of WizardGUI

/**
 * The action event handler for the checkboxes, and buttons.
 *
 * @param event Object that contains all the event information.
 * @param arg The object that the event is associated with.
 *
 */

public boolean action(Event event, Object arg)
{
    // Check for a check box action
    if (event.target instanceof Checkbox)
    {
        // determine if the forward button needs to be
        // disabled. This will only occur if both the
        // supplier and consumer check boxes are not selected.
        if (!consumerCB.getState() && !supplierCB.getState())
        {
            forwardB.disable();
        }
        else
        {
            forwardB.enable();
        }
    }

    // get the label associated with the check box
    String label = new String();
    label = ((Checkbox)event.target).getLabel();
    boolean state = ((Checkbox)event.target).getState();

    // determine if we are either the Consumer or Supplier check box
    if(label.equals("Consumer Application"))
    {
        consumerHideShow(state);
    }
    else if(label.equals("Supplier Application "))
    {
        supplierHideShow(state);
    }
}

```



```

    }

    // determine if we are going to generate a Push or Pull
    // communication model.
    if (consumerPushCB.getState())
    {
        consumerTF.setText("ConsumerPush.java");
    }
    else if (consumerPullCB.getState())
    {
        consumerTF.setText("ConsumerPull.java");
    }
    if (supplierPushCB.getState())
    {
        supplierTF.setText("SupplierPush.java");
    }
    else if (supplierPullCB.getState())
    {
        supplierTF.setText("SupplierPull.java");
    }
}

// determine which of the buttons have been pressed
if ("<< Back".equals(arg))
{
    cl.previous(cardStack);
    if (card_1.isShowing())
    {
        backB.disable();
    }
    finishB.disable();
    forwardB.enable();
}
else if ("Forward >>".equals(arg))
{
    cl.next(cardStack);
    // we are not displaying the first card so
    // enable the Back button to go back to the
    // previous screen
    if (! card_1.isShowing())
    {
        backB.enable();
    }
    // test for card 4
    if (card_4.isShowing())
    { // card 4 is being shown, disable the Forward button
      // if there is no channel name. Only allowed to proceed
      // to the next screen once a name for the channel has
      // been specified.
        if (channelTF.getText().length() == 0)
        {
            forwardB.disable();
        }
    }
    // the final screen is being displayed, enable/disable the
    // appropriate buttons and add the summary information to the
    // text area.
    if (card_5.isShowing())
    {
        finishB.enable();
        forwardB.disable();
        displaySummaryInfo();
    }
    else
    { // not on the last screen so disable the the Finish button
        finishB.disable();
    }
}
else if ("Finish".equals(arg))
{
    // generate the supplier and consumer applications
    generateCode();
    System.exit(0);
}
else if ("Cancel".equals(arg))
{ // exit the application, changes are lost.
    System.exit(0);
}
return true;
} // End of action

/**
 * Handles all other events that are not associated with an action
 * Indicates when a character has been entered into the Channel Name

```

```

* text field and sets the buttons accordingly.
*
* @param event Object that contains all the event information.
*
*/

public boolean handleEvent(Event event)
{

    if(event.id == Event.WINDOW_DESTROY)
    {
        System.exit(0);
    }

    // catch events for the text field. If displaying screen 4
    // and the user has entered a channel name then enable the
    // Forward button to proceed to the next screen. With no
    // channel name entered the Forward button will always be
    // disabled, the user will be unable to proceed.
    if (event.target instanceof TextField)
    {
        if (card_4.isShowing())
        {
            if (channelTF.getText().length() != 0)
            {
                if (! forwardB.isEnabled())
                {
                    forwardB.enable();
                }
            }
            else
            {
                if (forwardB.isEnabled())
                {
                    forwardB.disable();
                }
            }
        }
    }

    return super.handleEvent(event);
} // End of handleEvent

/**
 *
 * The first screen of the wizard application. Allows the user
 * to select the type of applications it is going to create
 * (either/or Supplier/Consumer).
 *
 */

private void CreateScreen1()
{
    // Create the Panel for the first screen (card)
    // in the card stack.
    card_1 = new Panel();

    // Create a Panel to hold the box object
    Panel panelBox = new Panel();
    Box selectBox = new Box(panelBox,"Select Application Type");

    // Associate a border layout with the first card
    card_1.setLayout(new BorderLayout());
    card_1.add("North", selectBox);

    // Associate a grid layout with the box Panel and
    // add the appropriate componets.
    panelBox.setLayout(new BorderLayout());
    panelBox.add("North",new Label("Select the type of application(s) you wish to create"));

    // Create another panel, and add components to the
    // Grid that is associate with the panel
    Panel compPanel = new Panel();

    consumerCB = new Checkbox("Consumer Application");
    supplierCB = new Checkbox("Supplier Application ");
    consumerCB.setState(true);
    supplierCB.setState(true);

    GridBagConstraints gbc = new GridBagConstraints();
    GridBagLayout gbl = new GridBagLayout();

    compPanel.setLayout(gbl);
    gbc.anchor = GridBagConstraints.NORTH;
    gbc.gridwidth = GridBagConstraints.REMAINDER;

```

```

gbl.setConstraints(consumerCB, gbc);
gbl.setConstraints(supplierCB, gbc);
compPanel.add(consumerCB);
compPanel.add(supplierCB);

panelBox.add("South", compPanel);

// Label this card as "screen1" and add to card stack
cardStack.add("screen1", card_1);

} // End of CreateScreen1

/**
 *
 * The second screen of the wizard application. Allows the user
 * to select the communication modle for the supplier/consumer
 * applications.
 *
 */

private void CreateScreen2()
{
// Create the Panel for the second screen (card)
// in the card stack.
card_2 = new Panel();

// Create a Panel to hold the box object
Panel panelBox = new Panel();
Box commBox = new Box(panelBox,"Communication Model");

// Associate a border layout with the card
card_2.setLayout(new BorderLayout());
card_2.add("North", commBox);

// Create a border layout for the comm panel and add the
// descriptive label to it.
panelBox.setLayout(new BorderLayout());
panelBox.add("North",new Label("Select the required communication models:"));

// Create the panel to hold the components
Panel compPanel = new Panel();
CheckboxGroup conGroup = new CheckboxGroup();
CheckboxGroup supGroup = new CheckboxGroup();

// Display components in a grid layout
compPanel.setLayout(new GridLayout(2,3));
consumerL2 = new Label("Consumer");
supplierL2 = new Label("Supplier");
consumerL2.setAlignment(consumerL2.CENTER);
supplierL2.setAlignment(supplierL2.CENTER);

consumerPushCB = new Checkbox("Push", conGroup, true);
consumerPullCB = new Checkbox("Pull", conGroup, false);
supplierPushCB = new Checkbox("Push", supGroup, true);
supplierPullCB = new Checkbox("Pull", supGroup, false);
compPanel.add(consumerL2);
compPanel.add(consumerPushCB);
compPanel.add(consumerPullCB);
compPanel.add(supplierL2);
compPanel.add(supplierPushCB);
compPanel.add(supplierPullCB);

// Add the componet panel to the comm panel
panelBox.add("South", compPanel);

// Add the second card panel to the card stack
// with the label "screen2" as the identifier
cardStack.add("screen2", card_2);

} // End of CreateScreen2

/**
 *
 * The third screen of the wizard application. Allows the user
 * to enter the names of the supplier/consumer applications that
 * will get created.
 *
 */

private void CreateScreen3()
{
// Create the Panel for the third screen (card)
// in the card stack.
card_3 = new Panel();

```

```

// Create a Panel to hold the box object
Panel panelBox = new Panel();
Box nameBox = new Box(panelBox,"File Names");

// Associate a border layout with the card
card_3.setLayout(new BorderLayout());
card_3.add("North", nameBox);

// Create a border layout for the name panel and
// add the label telling the user what to do.
panelBox.setLayout(new BorderLayout());
panelBox.add("North",new Label("Enter the name of the Supplier/Consumer files:"));

// Create another panel, and add components to the
// Grid that is associate with the panel
Panel compPanel = new Panel();
compPanel.setLayout(new GridLayout(2,2));

consumerTF = new TextField("ConsumerPush.java");
supplierTF = new TextField("SupplierPush.java");
consumerL1 = new Label("Consumer");
consumerL1.setAlignment(consumerL1.CENTER);
supplierL1 = new Label("Supplier");
supplierL1.setAlignment(supplierL1.CENTER);
compPanel.add(consumerL1);
compPanel.add(consumerTF);
compPanel.add(supplierL1);
compPanel.add(supplierTF);

// Add the panel with the components to the name panel
panelBox.add("South", compPanel);

// Associate the name "screen3" with this card and add
// to the card stack.
cardStack.add("screen3", card_3);
} // End of CreateScreen3

/**
 *
 * The forth screen of the wizard application. Allows the user
 * to enter the channel information, event channel name and
 * buffer size.
 *
 */
private void CreateScreen4()
{
// Create the Panel for the fourth screen (card)
// in the card stack.
card_4 = new Panel();

// Create a panel for the box object
Panel panelBox = new Panel();
Box channelBox = new Box(panelBox,"Event Channel");

// Create a border layout for the card, and add the box
card_4.setLayout(new BorderLayout());
card_4.add("North", channelBox);

// Create a border layout for the channel panel
panelBox.setLayout(new BorderLayout());
panelBox.add("North",new Label("Enter the Event Channel information:"));

// Create a panel to hold the components, display
// them in a grid layout.
Panel compPanel = new Panel();
compPanel.setLayout(new GridLayout(2,2));
Label channelName = new Label("Channel Name");
channelName.setAlignment(channelName.CENTER);
compPanel.add(channelName);
channelTF = new TextField();
bufferTF = new TextField("1000");
compPanel.add(channelTF);
Label bufferSize = new Label("Buffer Size");
bufferSize.setAlignment(bufferSize.CENTER);
compPanel.add(bufferSize);
compPanel.add(bufferTF);

// Add the component panel to the channel pannel
panelBox.add("South", compPanel);

// Add the card to the stack, with identifier "screen4"

```

```

cardStack.add("screen4", card_4);

} // End of CreateScreen4

/**
 *
 * The final screen of the wizard application. Simply displays
 * a summary of the information, selecting the Finish button
 * will generate the the Supplier/Consumer applications.
 *
 */

private void CreateScreen5()
{

    // Create a panel for the fifth card in the stack
    card_5 = new Panel();

    // Create a panel for the box object
    Panel panelBox = new Panel();
    Box summaryBox = new Box(panelBox,"Summary Information");

    // Associate a border layout for the card and add the box
    card_5.setLayout(new BorderLayout());
    card_5.add("North", summaryBox);

    // Add the component to the box panel
    panelBox.setLayout(new GridLayout(1,1));
    summaryTA = new TextArea(4,20);
    summaryTA.setEditable(false);
    panelBox.add(summaryTA);

    // Add the card to the stack, with identifier "screen5"
    cardStack.add("screen5", card_5);

} // End of CreateScreen5

/**
 *
 * Creates the buttons that allows the user to navigate through
 * the different screens, and to generate the required applications.
 *
 */

private void CreateButtons()
{
    // Create panel for the buttons
    Panel buttonBar = new Panel();

    // Use a flow layout
    buttonBar.setLayout(new FlowLayout());

    // Create and add the buttons
    backB = new Button("<< Back");
    forwardB = new Button("Forward >>");
    finishB = new Button("Finish");
    cancelB = new Button("Cancel");
    buttonBar.add(backB);
    buttonBar.add(forwardB);
    buttonBar.add(finishB);
    buttonBar.add(cancelB);

    // Add the button bar to the south part of the
    // border layout that is associated with the
    // main frame
    add("South", buttonBar);

} // End of CreateButtons

/**
 *
 * Depending on the state of the Consumer check boxes of the first
 * screen (i.e. if the box is selected or not) will determine if
 * the Consumer information appears in the other screens. For example,
 * if the Consumer check box is not selected then the information
 * specific to the consumer application will not appear in the other
 * screens. The wizard application will only generate the code for
 * the supplier application.
 *
 */

private void consumerHideShow(boolean state)
{
    // consumer application check box selected, show all
    // consumer specific objects in the latter screens
    if (state)

```

```

{
    consumerTF.show();
    consumerL1.show();
    consumerPushCB.show();
    consumerPullCB.show();
    consumerL2.show();
}
else
{ // consumer application check box not selected, hide the
// information.
    consumerTF.hide();
    consumerL1.hide();
    consumerPushCB.hide();
    consumerPullCB.hide();
    consumerL2.hide();
}
} // End of consumerHideShow

/**
 *
 * Depending on the state of the Supplier check box of the first
 * screen (i.e. if the box is selected or not) will determine if
 * the Supplier information appears in the other screens. For
 * example, if the Supplier check box is not selected then the
 * information specific to the supplier application will not
 * appear in the other screens. The wizard application will only
 * generate the code for the consumer application.
 */

private void supplierHideShow(boolean state)
{
    // supplier check box selected, display all supplier information
    // in the other screens.
    if (state)
    {
        supplierTF.show();
        supplierL1.show();
        supplierPushCB.show();
        supplierPullCB.show();
        supplierL2.show();
    }
    else
    { // supplier check box not selected, don't display any supplier
    // information in the other screens.
        supplierTF.hide();
        supplierL1.hide();
        supplierPushCB.hide();
        supplierPullCB.hide();
        supplierL2.hide();
    }
} // End of supplierHideShow

/**
 *
 * Displays a summary about the Consumer, Supplier and Event Channel
 * for the final screen.
 */

private void displaySummaryInfo()
{
    // remove any information already in the text area.
    summaryTA.setText("");

    if (consumerCB.getState())
    {
        String consumerModel = new String();
        summaryTA.append("Consumer \n");
        consumerFileName = consumerTF.getText();
        if (consumerFileName.length() == 0)
        {
            consumerFileName = "Consumer.java";
        }
        summaryTA.append("  File Name : " + consumerFileName + "\n");
        if (consumerPushCB.getState())
        {
            consumerModel = "Push";
        }
        else if (consumerPullCB.getState())
        {
            consumerModel = "Pull";
        }
        summaryTA.append("  Communication Model : " + consumerModel + "\n");
    }
}

```

```

}

if (supplierCB.getState())
{
    String supplierModel = new String();
    summaryTA.append("Supplier \n");
    supplierFileName = supplierTF.getText();
    if (supplierFileName.length() == 0)
    {
        supplierFileName = "Supplier.java";
    }
    summaryTA.append("  File Name : " + supplierFileName + "\n");
    if (supplierPushCB.getState())
    {
        supplierModel = "Push";
    }
    else if (supplierPullCB.getState())
    {
        supplierModel = "Pull";
    }
    summaryTA.append("  Communication Model : " + supplierModel + "\n");
}
summaryTA.append("Event Channel \n");
summaryTA.append("  Channel Name : " + channelTF.getText() + "\n");
summaryTA.append("  Buffer Size : " + bufferTF.getText() + "\n");

// set the text area screen to the top position
summaryTA.setCaretPosition(0);

} // End of displaySummaryInfo

/**
 *
 * Generates the code for either or both the supplier and consumer
 * applications depending on the user selections.
 *
 * @see TCDEvents.GUITools.FileOutput
 *
 */

private void generateCode()
{
    boolean consumerState = consumerCB.getState();
    boolean supplierState = supplierCB.getState();
    boolean supplierPush = supplierPushCB.getState();
    boolean consumerPush = consumerPushCB.getState();

    String channelName = channelTF.getText();
    String bufferSize = bufferTF.getText();

    if ( (consumerState) && (!supplierState) )
    { // generate consumer application only
        try
        {
            FileOutput gen = new FileOutput(consumerFileName, true,
                supplierPush, consumerPush,
                channelName, bufferSize);
        }
        catch(Exception e)
        { // no need to print out message already handled by the
            // FileCreateException class
            return;
        }
    }
    else if ( (!consumerState) && (supplierState) )
    { // generate supplier application only
        try
        {
            FileOutput gen = new FileOutput(supplierFileName, false,
                supplierPush, consumerPush,
                channelName, bufferSize);
        }
        catch(Exception e)
        {
            return;
        }
    }
    else if ( (consumerState) && (supplierState) )
    { // generate both supplier and consumer applications
        try
        {
            FileOutput gen = new FileOutput(consumerFileName, supplierFileName,
                supplierPush, consumerPush,
                channelName, bufferSize);
        }
    }
}

```

```
        catch(Exception e)
        {
            return;
        }
    }
} // End of generateCode
} // End of class WizardGUI
```