Design and Implementation of CORBA-Based Subscription Server[•]

Ritika Maheshwari & Rod Fatoohi
College of Engineering, San Jose State University, San Jose, California 95192, USA.
Email: rfatoohi@email.sjsu.edu

Abstract

The standard CORBA operation invocation model supports synchronous, one-way, and deferred synchronous interactions between clients and servers. However, this model is too restrictive for real-time applications. In particular, the model lacks asynchronous message delivery, does not support timed invocations or group communication, and can lead to excessive polling by clients.

In this paper, we have designed and implemented a subscription server based on the OMG Notification Service specification for the structured push style communication. In order to alleviate some of the restrictions with the standard CORBA invocation model, the Subscription Server supports asynchronous message delivery and allows one or more suppliers to send structured messages to one or more consumers. Event data can be delivered from suppliers to consumers without requiring these participants to know about each other explicitly.

In addition to implementing most of the interfaces defined in the specification, the Subscription Server implements an additional advertiser interface, which lets one discover all event types offered by the channel and subscribed by the consumers. Event filtering as well as quality of service properties, such as connection reliability and event reliability, are also provided. An Auction Alert application was developed to test the functionality of the subscription server.

1. Introduction

There is a growing interest in the use of general-purpose event notification services as "middleware" for gluing together independently developed distributed applications. Distributed systems are intended to form the backbone of emerging next-generation communication systems, including electronic commerce, PCS, satellite surveillance systems, distributed medical imaging, real time data feeds, and flight reservation systems. One obvious benefit of distributed systems is that they reflect the global business and social environment in which we live and work.

From a communication point of view, The Common Object Request Broker Architecture (CORBA) [12] allows for synchronous, one-way and deferred synchronous requests. With synchronous requests, the client application is blocked until the request is returned to the client. With one-way requests, the client sends a request and proceeds with other work without checking for a response. Here the receiver does not return a value to the sender. With a deferred synchronous request no blocking occurs. A client application that issued the request is not blocked and is free to do whatever it likes. However, it must periodically poll the ORB for a response from the object, which executes the request.

[•] Research was partially funded through the Cooperative Agreement No. NCC2-1071 between NASA Ames and San Jose State University.

In principle, synchronous invocations simplify the development of distributed applications by supporting an implicit request/response protocol that makes remote operation invocations transparent to the client. In practice, however, the standard CORBA operation invocation model is too restrictive for real-time applications. Moreover, standard one-way invocations might not implement reliable delivery and deferred synchronous invocations require the use of the CORBA dynamic invocation interface, which yields excessive overhead for most real-time applications.

There is no provision in the CORBA core for features such as asynchronous event-driven communication. The CORBA core addresses only requests where the network connection between the client object and the server object must be maintained. This type of communication can support only tightly coupled clients and servers. This does not necessarily reflect the reality of the business activities, which often evolve unpredictably and asynchronously. To be more viable, the CORBA architecture should support connectionless and event-driven communication. It should have an ability to de-couple among client objects and server objects [15].

In this paper, we look at several CORBA-based asynchronous communication solutions. First, we discuss the Event Service (section 2), followed by a description of the Notification Service (section 3). Then we describe in detail our Subscription Service, which is based on the Notification Service (section 4). An auction alert application that uses the Subscription Server is briefly described afterward (section 5). Finally, we present some concluding remarks.

A subscription server represents an important component of the NASA Earth Observing System Data Information System (EOSDIS) core system (ECS) [14]. The EOSDIS is one of the largest information systems ever built that consists of legacy systems and data, variety of computers, high-speed networks and satellites. The current design of the ECS is built on the Distributed Computing Environment (DCE) [12] by the Open Group with an object-oriented wrapper called OODCE. NASA is interested in migrating into a CORBA-based middleware. One of the main goals of this project is to design a subscription server that can replace the EOSDIS Subscription Server but based on CORBA. The functionality of our subscription server mimics the EOSDIS one, yet our server is general enough to be used for other applications as we demonstrate its applicability to an auction alert system. In this paper, we also outline the similarity between the EOSDIS Subscription Server and our subscription server (section 6).

2. Event Service

As a solution to the problems with the CORBA invocation model, the Object Management Group (OMG) came up with the Event Service specification [7]. The Event Service provides an application architecture where there are no clients or servers. There are suppliers (or generators) of information and consumers of information. The Event Service de-couples communication from specific client-server pairs. The heart of the OMG Event Service is the ***event channel***, which is an object that does the actual de-coupling between suppliers and consumers. Alone, the event channel is the consumer to the suppliers and the supplier to the consumers. The event channel accepts connections from one or many suppliers, and one or many consumers. The key is that any event received from one supplier is transmitted to every consumer, as shown in Figure 1 [6]. Furthermore, multiple event channels are supported, working independently of each other.

The Event Service supports two models of operation: a push model and a pull model. Both models govern how suppliers and consumers communicate with an event channel. In a push model, a push supplier sends an event to the event channel using a CORBA push operation. This is unsolicited, event-driven processing to the event channel. In a pull model, the event channel,

acting as a client to the suppliers, polls the suppliers for information. Such pulling can be based on time intervals. On the consumer side, both push and pull models are supported.

Communication between the event channel and each supplier and consumer consumes a network connection. Thus there are clear limits to scalability and the number of concurrent connections each event channel can handle. In fact, when an event from a supplier needs to be distributed to many consumers, an event channel will send it to each consumer connected to it. Such operations are not efficient and can consume substantial network bandwidth.

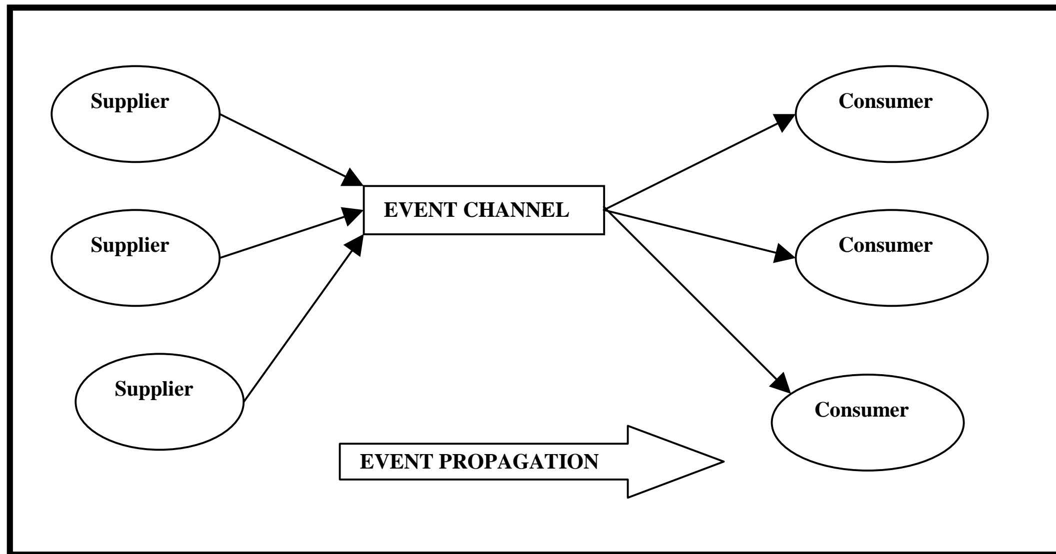

Figure 1 Communication through Event Channel

The Event Service also raises additional questions. What happens to the event if the connection between the event channel and either the supplier or the consumer fails in the middle of transmission? Can one-time, and only one-time, delivery of an event be guaranteed? If an event channel has the capacity to hold an event's information, how long should it be held and how, and who will control it? Are all events equal and is delivery indiscriminate, or do we discriminate events by a priority that governs the order that events will be sent to the ultimate consumers?

3. Notification Service

The answer to most of these questions lies in the Notification Service specified by the OMG [9]. The Notification Service incorporates all the basic features of event communication and is fully compatible with the Event Service. However, the Notification Service also provides many additional and powerful features that enable the implementation of highly sophisticated, intelligent event-based communication. The Notification Service has the following main features:

- **Event Filtering:** It can filter events and deliver only certain types of events to a consumer unlike the event service, which delivers all events to all consumers.
- **Quality of Service (QoS):** The characteristics of event such as priority and reliability can be set, which was not possible in the Event Service.
- **Structured Event**: Events can also be well-defined data structures, in addition to CORBA *Any* supported by the existing Event Service (A CORBA *Any* is a data type that can hold any primitive or user-defined CORBA type).

- **Event Subscription**: Consumers can indicate what type of event they want to receive, so that suppliers can produce events on demand, or avoid transmitting events in which no consumer has interest.
- **Event Offers:** Suppliers to the event channel can indicate what types of event they can supply so that the consumers can subscribe to these types as they become available.
- **Health Check:** It provides a health check facility that enables both the client and channel to monitor the viability of their connection to one another.

Notification is an efficient, intelligent means of distributing information to many recipients in large network environments. Suppliers of information publish new events to the notification service. Consumer of information subscribes to the notifications needed and the service redistributes information as required. A key advantage of this approach is that both information suppliers and consumers can be developed separately without intimate knowledge of each other. For example, new information suppliers and consumers can be incorporated without any changes whatsoever for existing information suppliers or other information consumers.

The Notification Service is sometimes confused with Message Queuing [12], which is a middleware technology that fits somewhere between RPC/RMI and the Notification Service. Message queues implement virtual mailboxes (or queues) where producers can send mail, and consumers can later pick it up. This asynchronous style of communication is superficially similar to notification. Unlike notification however, message queues are named, and the sending application must explicitly direct messages to a queue, which knows that a receiver is going to check. Notification uses a different mechanism, relying on the service to decide where to send notifications. Producer applications do not need to know what happens to their notifications once they are sent. Message queuing and notification are two different styles of communication, and while initially they look similar, a closer analysis will determine which one is needed.

CORBA defines the Notification Service at a level above the ORB architecture [9]. There are several key concepts in the Notification Service, summarized below.

- **Event** is a specific instance of an event message about a particular event. For example an event message may be generated to report that an insurance agent has closed a contract for an insurance policy. This event may be monitored by one program that maintains statistics on the number of contracts that are closed on average per hour, and by another program that logs all activities of each agent. Event messages are discrete, relating only to one event. However, events can be replicated to report the same event information to multiple consumers.
- **Event Supplier** is an object that generates event messages; it is a supplier of events, or more literally, the messages that report that event.
- **Event Consumer** is an object that receives event messages making it a consumer of events.
- **Event Channel** (also here called the notification channel) is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. It provides IDL interfaces, to bind both consumers and suppliers to proxy objects within the channel, and administrative interfaces, to specify various QoS and administrative properties that will be supported by the channel. The standard administrative properties that can be set on a channel include the maximum number of events the channel can buffer at any given time, *MaxQueueLength*, and the maximum number of consumers and suppliers that can connect to the channel, *MaxConsumers* and *MaxSuppliers* respectively.
- **Proxy Objects** are objects used by the event channel to bind consumers and suppliers. The channel creates one proxy object for each supplier or consumer. For example, when a

PushSupplier object registers with the channel, the channel creates a corresponding *ProxyPushConsumer* object. When *PushSupplier* needs to push an event onto the channel, it does this by invoking an operation on *ProxyPushConsumer*. The Notification Service provides separate client and proxy interfaces depending on the desired form of message communication. There are four basic kinds of events in the Notification Service: events of type *Any*, events in the form of structured events, events in the form of sequences of structured events, and events in the form of strings. When an application is passing information which does not fit into a particular structure and the information structure may vary from time to time, then events of type *Any* are used. When highly structured information is being passed in an application like an insurance application, where a person's insurance related data is being passed, structured events are used. In applications where a group of structured events have to be passed, it is more efficient to pass events as a sequence of structured events rather than an individual structured event. For example an inventory system, which sends the report regarding the inventory status for various departments in the store at the end of the day, would probably use sequence of structured events. The events in form of strings are used when they have to pass quickly through the notification channel, and do not require QoS properties and filtering.

- **Admin Object** is a factory that creates the proxy objects to which consumers and suppliers ultimately connect. Each event channel can support several objects born from the *ConsumerAdmin* and *SupplierAdmin* interfaces. The Notification Service treats each admin object as the manager of a group of proxies it has created. Admin objects can themselves have QoS properties and filter objects associated with them. The set of filter objects associated with a given admin is treated as a unit where these filters apply at all times to all proxy objects created by the admin. Additional filter objects can be associated with an individual proxy, but the set of filter objects associated with an admin object are automatically associated with all proxy objects, which have been created by the admin object.
- **Filter Objects** control the events that a consumer object receives by encapsulating a set of constraints specified in a particular constraint grammar within the filter objects. Each admin and proxy object within the event channel can therefore have one or more filter objects associated with it. There are two types of filter objects: *event-forwarding filters,* which affect event-forwarding decisions made by proxy objects, and *mapping filters*, which affect the way a proxy object treats events with respect to certain QoS properties. Each event has two special properties that can influence the delivery policy applied to the event: its priority and its expiration time. Although the supplier often populates these properties, a consumer's opinion of the relative importance of the event can often differ from that of the supplier. Mapping Filter objects allow consumers to affect the priority and lifetime properties of events.
- **QoS Properties** allow the tuning of the Notification Service according to specific priorities. These properties include reliability, priority, expiry time, earliest delivery time, order policy, discard policy, maximum batch size and pacing interval. The QoS properties can be set at the event channel, supplier admin, consumer admin, proxy suppliers, proxy consumers and individual event messages.

There are a number of products in the market based on the Notification Service such as dCon [2] by DSTC (Distributed Systems Technology Center) which provides persistent connection, persistent event reliability, push and pull event models, *StartTime* and *StopTime*, and all OMG-defined QoS properties and forwarding and mapping filters. Other products include ORBacus Notify by OOC (Object Oriented Concepts) [11], OpenFusion by Prismtech [13], and OrbixNotification by Iona [6]. Iona has also OrbixTalk, which is based on the Event Service with some QoS properties.

4. Subscription Service

The designed Subscription Server is a subset of the OMG Notification Service specification [9]. The main features supported by the server are: push type communication model; structured events; event forwarding filters; QoS properties such as connection reliability and event reliability; admin properties such as *MaximumSuppliers*, *MaximumConsumers* and *MaximumQueueLength;* in-memory persistent; out-of-memory restoration; and an advertiser. The advertiser, which is similar to the advertiser in the EOSDIS Subscription Server, advertises the offers made by the suppliers of the channel and the subscriptions made by the consumers. Basically our Subscription Server mimics the EOSDIS Subscription Server. We based our design on the OMG Notification Service, instead of the OMG Event Service for example, since the Notification Service is more general and is closer to the EOSDIS one than the Event Service.

The designed Subscription Server implements most of the functionality encapsulated in the structured push style communication interfaces. However, it implements only event forwarding filters, whereas the Notification Service specifies both event forwarding and mapping filters. In the Subscription Server the filter constraints are written in the Subscription Server Filter Constraint Language, which is a scale down version of the Default Filter Constraint Language specified by the Notification Service. Besides connection reliability and event reliability, other QoS properties specified by the Notification Service - such as priority, event earliest delivery time, expiry time and maximum batch size - were not implemented. Moreover, the advertiser interface implemented in the Subscription Server is not part of the Notification Service IDL.

4.1. Object Diagram and Event Flow

The architecture and the event flow of the Subscription Server are summarized in Figure 2. The event channel is created by an *EventChannelFactory* object. This factory object as well as the event channel has database objects connected to them, to make them persistent. A single factory object can create multiple event channels. There can be multiple *SupplierAdmin* and *ConsumerAdmin* objects attached to the event channel. Each *SupplierAdmin* object can have multiple *StructuredProxyPushConsumer* objects attached to it. There can be zero or more filter objects attached to *SupplierAdmin*, which has a database object. Each *Supplier* object is connected to a *StructuredProxyPushConsumer* object, which has a database object and zero or more filter objects attached to it. A similar even flow occurs at the consumer side.

The supplier pushes the structured event to its *StructuredProxyPushConsumer* object. If the event passes the filters attached to the proxy object, it is passed to the *SupplierAdmin* object. If the event passes the filters attached to *SupplierAdmin*, then it is passed to all *ConsumerAdmin* objects attached to the event channel. Similar filtering operations are performed at the consumer side.

4.2. Structured Events

Each structured event comprises of two main components: a header and a body [6]. The header is further divided into a fixed part and a variable part. The fixed part of the header consists of three-string fields: domain name*,* event type and event name*.* The variable part of the header consists of a list of zero or more name-value pairs, where each name is a string and each value is an *Any*. These fields are optional and their contents are virtually unbounded. However there is a standard set of well-defined optional header field names and defined data types for their value.

The event body is further divided into a filterable portion and the remainder of the body. The filterable body fields contain the most interesting fields of the event, upon which the consumer is most likely to base filtering decisions. Like the optional header fields, the filterable portion of the event body is also defined as a sequence of name-value pairs, with each name being a string and each value an *Any*. The remainder of the body, defined as an *Any*, provides a convenient place to transmit any event data in addition to that defined in the filterable body fields.

4.3. Event Forwarding Filters

These filters affect the event forwarding decisions made by the proxy objects. They encapsulate a set of constraints. Each constraint is a data structure with two components: a sequence of data structures - each of which indicates an event type - and a string containing a Boolean expression - whose syntax conforms to the Constraint language for the Subscription Server.

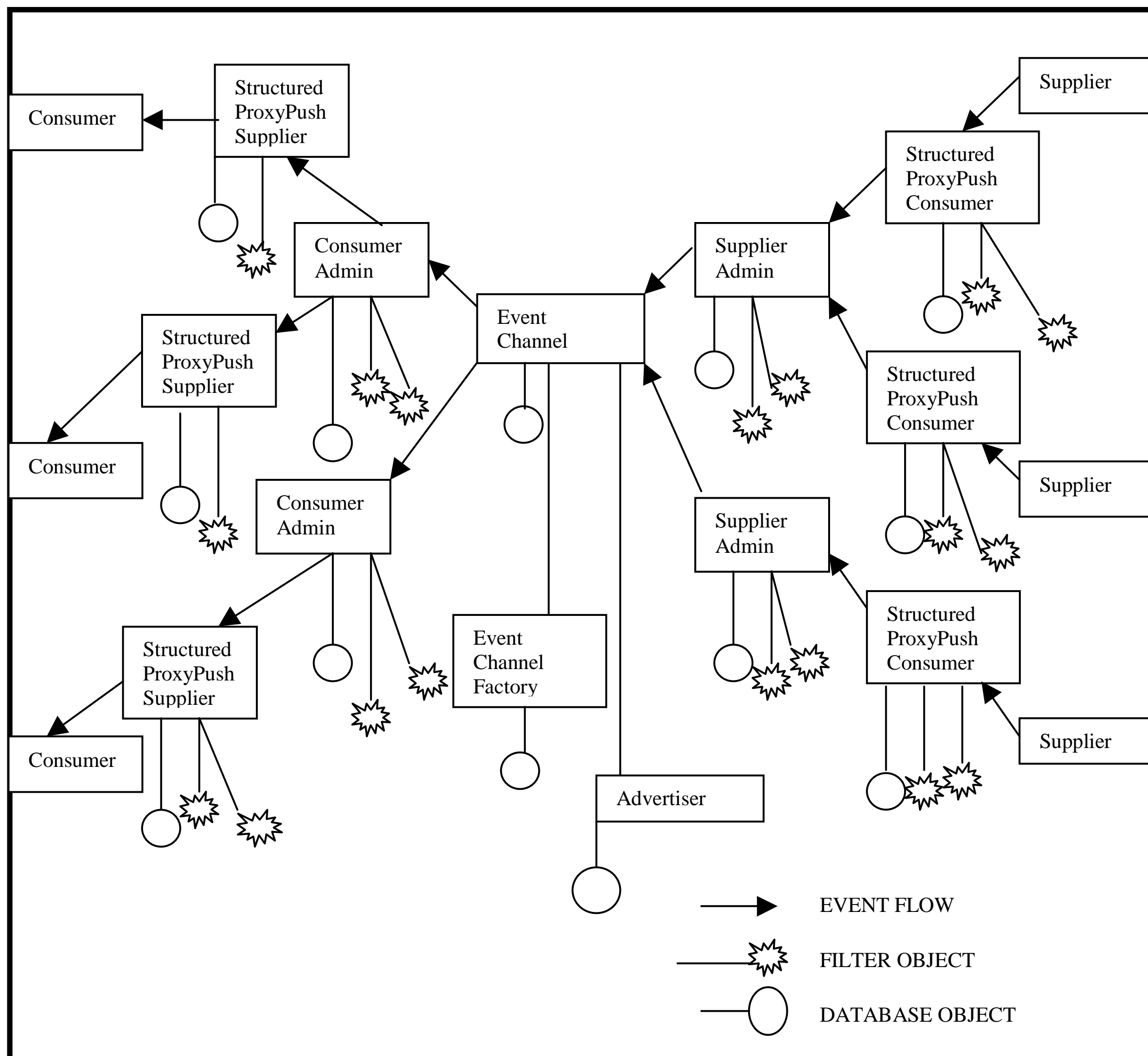

Figure 2: Subscription Server Architecture.

Filters can be associated with either proxy objects and/or admin interfaces. When a proxy object within an event channel receives an event, it invokes an appropriate match operation on each of

its associated filter objects. A match operation accepts as input the contents of the event being filtered against and returns a Boolean result. The result is TRUE if the event satisfies one or more of the constraints encapsulated by the filter object (that is, OR semantics are applied between the constraints encapsulated by a filter object), and FALSE otherwise. If the proxy has multiple filter objects associated with it, it invokes the match operation on each of its associated filter objects until either one returns TRUE, or all have returned FALSE.

A set of filter objects can also be associated with each admin interface within an event channel. The set of filter objects associated with an admin object thus applies to each proxy object associated with that admin, and can only be modified by invoking operations on the admin object, and any such modifications affect all the proxy objects under the management of that admin.

In addition, each proxy object can have two sets of filter objects: those that are associated with its managing admin objects and those that are added to it directly. Upon creation of each admin object, a flag can be set which indicates whether each proxy object created by the admin will AND or OR the results of applying these two sets of filter objects when determining whether or not to forward each event. Within each set, only OR semantics are applied in all cases; the flag only affects the operator used to combine the results of applying each of the two sets of filter objects to each event.

The parser and the lexical analyzer used to parse the constraints encapsulated by the filter object are JLex and CUP (available free-of-charge from Princeton University website). The JLex utility [1] is based upon the Lex lexical analyzer generator model. It takes a specification file similar to that accepted by Lex and creates a Java source file for the corresponding lexical analyzer. CUP [5] is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC and in fact offers most of the features of YACC. However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers, which are implemented in Java.

A proxy, who has no filters associated with it, sends through all events it receives. In the case of proxy consumers, all events are passed to the proxy suppliers on its channel, and in the case of proxy suppliers, all events are delivered to its connected consumer.

4.4. QoS Properties

The Subscription Server provides two major QoS properties namely event reliability and connection reliability. There are variety of delivery policies known in the distributed systems, such as best effort, at-least-once, at-most-once, and exactly-once. However most of these only make sense in a point-to-point, request–reply communication model. The Subscription Service is by definition a point-to-multipoint delivery mechanism with no explicit reply mechanism [6].

The Subscription Server treats the reliability of specific events and the reliability of the connections, which provide a transport for events between clients and the event channel, as separate issues and so it defines two separate QoS properties to represent them: event reliability and connection reliability. Each of these properties can take on one of two possible numeric constant values: best-effort or persistent. The QoS properties related to reliability can be set at the channel, group and proxy levels, but not for individual events.

The Subscription Server also supports the configuration of certain administrative properties, each of which has an associated value of type *long* on a channel. These properties are: *MaxQueueLength* - the maximum number of events that can be queued by the channel before the

channel begins discarding events upon the receipt of each new event, *MaxConsumers* - the maximum number of consumers that can be connected to the channel at any given time, and *MaxSuppliers* - the maximum number of suppliers that can be connected to the channel at any given time.

4.5. In-memory persistent objects

The state of all objects in the object hierarchy of the Subscription Server is kept in memory. This is done to enhance performance. Since the subscription service is a real time service, its very purpose would be defeated if the state of the channel objects was not kept in the memory, and would have to be fetched from the database every time an operation was invoked on the object. In other words having stateless objects would have a big effect on the performance of the Subscription Server.

Besides keeping the state of the objects in the memory, their state is also made persistent by storing it in the database. So in case there is a server crash, the event channel can restore its state before the crash. Therefore at any time when the state of an object changes, it calls an appropriate method on its database object, which in turn updates the state of the object in the database.

One major drawback of this system is that it does not swap out objects from the memory, depending on some swapping policy (like the least used object), so that at a given time only a limited number of objects are kept in the memory. This kind of swapping would make the system more scalable. But implementing a swapping policy would have been a major project in itself and therefore, it was not pursued.

4.6. Restoring the State of Subscription Server

Since the object state is made persistent at every level in the object hierarchy, it is possible to restore the state of the subscription server after the server crashes. The restore operation is a cascading operation, which cascades through the entire object hierarchy as it restores the state.

4.7. System Analysis

The hardware and software requirements for the development of the Subscription Server are: an Intel Pentium based computer with at least 90 MB RAM and 200 MHz speed; JDK 1.2.2; Inprise Visibroker 3.3 ORB and naming service; CUP, a LALR parser generator for Java, version 0.10; Jlex, lexical analyzer generator for Java, version 1.2; and DB2 Universal database version 6.1.

5. Auction Alert System: an Application of Subscription Server

In order to demonstrate the functionality of the Subscription Server, an auction alert application has been developed. In this application the consumers and the suppliers connect to the Subscription Server, as shown in Figure 3. The suppliers put the offers for the items they want to auction on the Subscription Server. The consumers register their subscriptions about the auction notification of the items they are interested in. This application is just a notification system and does not take care of the bidding part of the auction scenario.

6. Similarities between the EOSDIS and the Designed Subscription Servers

After going through the architecture of both the EOSDIS Subscription Server and our Subscription Server, the similarities between the two are very apparent. Table 1 lists these

similarities, point by point. Thus there is a very close functional similarity between the EOSDIS Subscription Server and the designed Subscription Server. Both servers basically support structured push style of communication. If a decision is made to port the EOSDIS infrastructure to CORBA, the designed Subscription Server could possibly substitute the EOSDIS Subscription Server. The porting to CORBA could be done by either using a CORBA/DCE bridge [3], since the current ECS infrastructure is OODCE based, or by remodeling the EOSDIS infrastructure to be CORBA based.

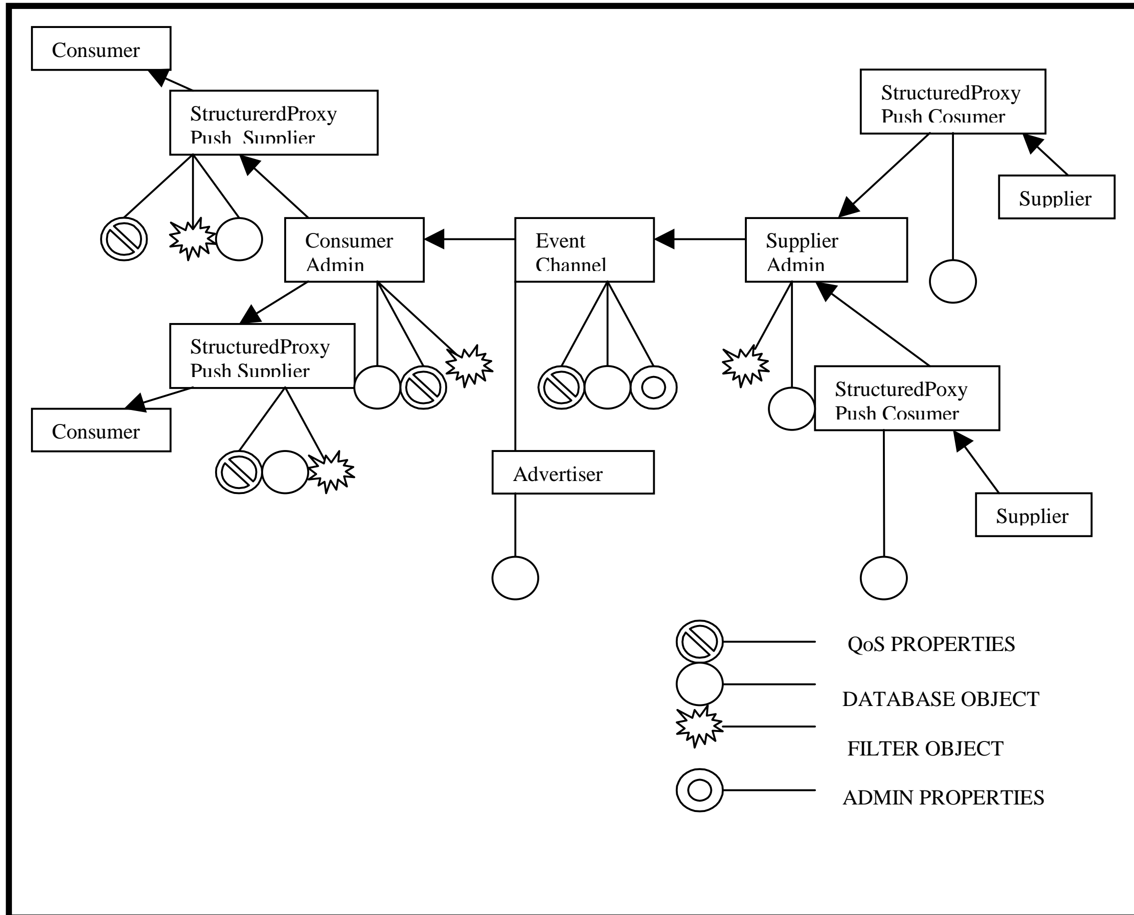

Figure 3: Auction Alert System Architecture

7. Conclusions

The OMG has successfully created standards for middleware infrastructure and is currently developing interoperability standards in many vertical domains based on the Object Management Architecture (OMA). In many environments today enterprises buy best of breed systems that need to communicate to each other. Using standard services specified in OMG IDL, systems from heterogeneous vendors can be deployed in an enterprise with minimal specialization for each environment. CORBA provides communications between the separate systems using synchronized operations between a client and server. Synchronized operations are applicable for commands, queries and other polled mode (or pull style) interaction [10].

Table 1. Similarities between EOSDIS and designed Subscription Servers.

	EOSDIS Subscription Server	Designed Subscription Server
1	It has an advertiser to advertise events when it receives event offers from the event providers. The clients discover the offered events from the advertiser	It has an advertiser to advertise events offered by the suppliers of the Subscription Server. The clients discover the events from the advertiser.
2	It supports previously defined events.	It supports structured events, which have a pre-defined structure.
3	The client interested in receiving events invokes submit subscription method on the server and submits the request. The request qualifiers are validated and a unique ID is returned to the client.	The client interested in receiving events connects to the server through a proxy supplier object, which has a unique ID. The client attaches filter objects to the proxy, in order to receive only those events in which it is interested. So, instead of validating the qualifiers, the client attaches filter objects to the proxy.
4	When the event provider triggers the event, all the clients whose subscription qualifiers match the event qualifiers send a notification and the pre-registered actions are invoked on the client's behalf	When the generated event passes the proxy and the admin filters, then the proxy supplier object invokes the *push_structured_event()* operation on the consumer. Here the consumer specifies the action, which needs to be carried out in case the particular event occurs.

Many applications require the support for asynchronous, event-based communication [4]. The OMG Notification Service defines capabilities that enhance the support for event-driven CORBA applications provided by the OMG Event Service. Specifically, the Notification Service enables clients connected to an individual event channel to subscribe to specific events based on their type and contents, and to configure an individual channel to support various QoS properties.

However the proposed Subscription Server has some drawbacks:

- The objects are kept in the memory for performance gain, but this becomes a problem as far as scalability goes. In order to be scalable, the Subscription Server should have some kind of life cycle management. It should be able to swap out the least used objects from the memory and keep the others (depending on some swapping policy) in memory. Ideally there should be some kind of an OTM (Object Transaction Monitor), which would activate and deactivate the objects and maintain their persistent state.
- Another drawback with the current architecture is security. Any application having the reference to an event channel can get the references to all its admin objects. From the admin objects, it can get references to all the proxy objects attached to an admin. Also, from the proxy object they can reach the supplier or the consumer (depending on the proxy). Thus it is imperative to come up with better security system in the current architecture.

A future endeavor could be to develop an Enterprise Java Beans (EJB) based Subscription Server, and let the EJB container take care of life cycle management.

In the end we would like to point out a few shortcomings in the current Notification Service specification, which can be fixed in the future enhancements to the specification.

- The current Notification Service specification provides a minimal amount of IDL and semantics for the end points (suppliers and consumers) to connect to the notification

channel. Additional IDL is typically developed by each application needs to connect to the notification channel. Therefore there is a need to standardize the additional set of capabilities that is common to many suppliers and consumers of event notification in order to provide higher degree of interoperability [10].

- One important requirement not covered by the Notification Service is the definition of standard management interfaces. While it is not difficult to conceive of management applications that use the interfaces defined by the Notification Service to configure, monitor and control event channels, these interfaces do not directly provide these higher-level management capabilities. Thus, defining higher-level management interfaces will enhance the interoperability of applications written to manage an implementation of the OMG Notification Service.
- In addition to defining management interfaces for configuring a single Notification Service event channel, its common for end users to want to chain together multiple channels, such that one channel consumes the events supplied by another. Such a configuration is important for event-driven applications deployed across wide area networks. In such a deployment, end-users require the ability to configure a network of chained channels to perform higher level, network-wide functions, and to provide capabilities such as fault tolerance and load balancing. Thus additional interfaces are required for creating and managing such event networks [8].

References

[1] Berk, “JLex: A Lexical Analyzer Generator for Java”, Manual. http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html

[2] DSTC, dCon. http://obsolete.dstc.edu.au/Products/CORBA/Notification_Service.

[3] Fatoohi, Gunwani, Wan, & Zheng, “Performance Evaluation of Middleware Bridging Technologies”, 2000 IEEE Int. Symp. On Performance Analysis of Systems & Software (April 2000, Austin, TX), pp. 34 - 39.

[4] Harrison, O’Ryan, Levine & Schmidt. “The Design and Performance of a Real-time CORBA Event Service”, In the Proceedings of the OOPSLA 97 Conf, Atlanta, Georgia, October 1997.

[5] Hudson, “CUP LALR Parser Generator for Java” Manual, http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html

[6] IONA Technologies, “OrbixNotification Programmers Guide and Reference”, http://www.iona.com/products/messaging/notification/developer.html

[7] Object Management Group, “Event Service Specification”, ftp://ftp.omg.org/pub/docs/formal/97-12-11.pdf

[8] Object Management Group, “Management of Event Networks RFP”, OMG Doc 98-06-17.

[9] Object Management Group, “Notification Service Specification”, OMG Doc 99-07-01. ftp://ftp.omg.org/pub/docs/telecom/99-07-01.pdf

[10] Object Management Group, “Publish/Subscribe RFP”, OMG Doc 98-03-05.

[11] Object Oriented Concepts, “ORBacusNotify”, http://tuvok.ooc.com/notify/.

[12] Orfali, Harkey, & Edwards, Client/Server Survival Guide, Wiley, 1999.

[13] PrismTech, “OpenFusion Notification Service”, 1999. http://www.prismtechnologies.com/

[14] Raytheon Systems, CSS overview. Release 4 Segment/Design Spec, 305-CD-100-005.

[15] Resnick, CORBA Async Requirements, http://www.interlog.com/~resnick/corbaasync.txt