A PERSISTENT AND RECOVERABLE
MIDDLEWARE APPROACH TO
ALERT DISTRIBUTION

The members of the Committee approve the masters thesis of Nishanth Reddy Vontela

Sharma Chakravarthy
Supervising Professor _____

Alp Aslandogan _____

Leonidas Fegaras _____

A PERSISTENT AND RECOVERABLE MIDDLEWARE APPROACH TO ALERT DISTRIBUTION

by

NISHANTH REDDY VONTELA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2002

To my friends, family and loved ones for their consistent support,
Encouragement and love

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his great guidance and support, and for giving me an opportunity to work on this project. I am also thankful to Dr. Leonidas Fegaras and Dr. Alp Aslandogan for serving on my committee.

I would like to thank Pratyush Mishra for maintaining a well-administered research environment and being so helpful at times of need. Thanks are due to all members of ITLAB for their invaluable help and fruitful discussions during the implementation of this work. Also I would like to thank all my friends for their support and encouragement.

This work was supported, in part, by the office of Naval Research, the SPAWAR System center-San Diego & by the Rome Laboratory (grant F30602-01-2-05430), and by NSF (grant IIS-0123730).

I also thank my parents and sister for their constant support and encouragement throughout my academic career.

March 29, 2002

ABSTRACT

A PERSISTENT AND RECOVERABLE

MIDDLEWARE APPROACH TO

ALERT DISTRIBUTION

Publication No._____

Nishanth Reddy Vontela, M.S.

The University of Texas at Arlington, 2002

Supervising Professor: Sharma Chakravarthy

Notification Systems attempt to provide a mechanism for signaling the occurrence of an event and informing interested parties through alerts or messages. Typically, they support a mechanism for dynamically registering an interest in some types of events, thus removing the necessity of establishing dependency relationships between event producers and consumers at build time. This selective delivery of notification is fundamental: conventional broadcast and multicast communications mechanisms fail in large-scale distributed environments due to the sheer number of recipients. A subscription-based service can drastically reduce the required fan out.

Decoupling the production and consumption of information in software systems facilitates extensibility by removing explicit dependencies between components. So called "publish/subscribe" notification architectures are comprised of undirected production, and subscription to events by their characteristics rather than their source.

Alert Server is such a notification service, being developed at ITLAB. This thesis discusses its design, implementation and use.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Enterprise messaging products (or as they are sometimes called, Message Oriented Middleware or MOM products) are becoming an essential component for integrating intra-company operations. They allow separate business components to be combined into a reliable, yet flexible, system. In addition to the traditional MOM vendors, several database vendors and a number of Internet related companies also provide enterprise-messaging products.

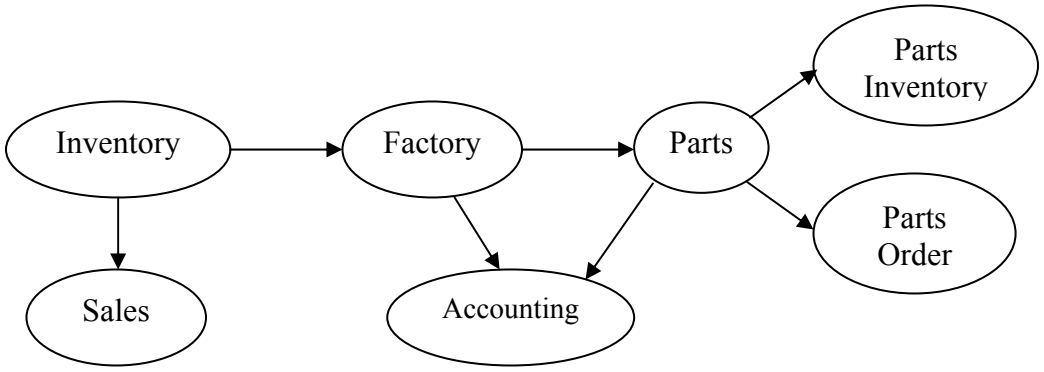


Figure 1.1. Messaging in an application.

MOMs have a lot of applications. For example, components of an enterprise application for an automobile manufacturer can use a message oriented middleware for the following:

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level, so the factory can make more cars.
- The factory component can send a message to the parts components so that the factory can assemble the parts it needs.
- The parts components in turn can send messages to their own inventory and order components to update their inventories and order new parts from suppliers.
- Both the factory and parts components can send messages to the accounting component to update their budget numbers.
- The business can publish updated catalog items to its sales force.

Using messaging for these tasks allows the different components to interact with each other efficiently, without tying up network or other resources. Manufacturing is only one example of how an enterprise can use messaging system and its API. This can also be used in financial services applications; health services applications and many other applications. Thus the basic aim in all these applications is the distribution of processing across multiple processors and platforms. These applications can be built using many alternative approaches. This thesis discusses the design and implementation of one such messaging system called Alert Server.

1.1 Problems and Requirements

In traditional network applications, when two processes must communicate with each other, they need network addresses to begin communicating. If a process wants to send a message to many other processes, it first would need to know the physical network addresses of the other processes and then create a connection to all those processes. This architecture does not scale well because configuration is complicated and tedious. More

over these messages can be delivered in any order and these processes have to keep track for what messages they are interested. These processes can be any client applications on the same host or different hosts on a network that can produce messages or consume messages or do both. This is shown in figure 1.2. It is in this scenario that Alert Server is needed where in different client applications can pass the alerts to the Alert Server and the server distributes the alerts to only those client applications that are interested in these alerts. In our case these client applications can be a Global Event Detector (GED) or Local Event Detector (LED) [1, 2]. GED and LED, developed at Sentinel, are used to provide active capabilities to traditional databases. The clients can generate events and send them either to LED for detection or to Alert Server to be just passed on to other interested clients. Apart from all these, it is also essential to build a system using openly available components and not to commit to any specific vendor.

This thesis discusses the design and implementation of a messaging system called Alert Server. There are several motivations behind our objective of designing and implementing a message distribution mechanism even though we have many existing messaging systems (like Java Message Queue (JMQ), IBM MQ Series etc). Firstly, not all messaging system implementations support all operating systems and protocols. Secondly, internal infrastructure of these systems cannot be modified to achieve our goals that are explained in the next sections. Thirdly, there is significant overhead incurred if we were to build the Alert Server on top of these existing systems, as we do not require all the functionalities supported by them. Also, most of the messaging systems do not support priority-based delivery of alerts and recovery from crashes.

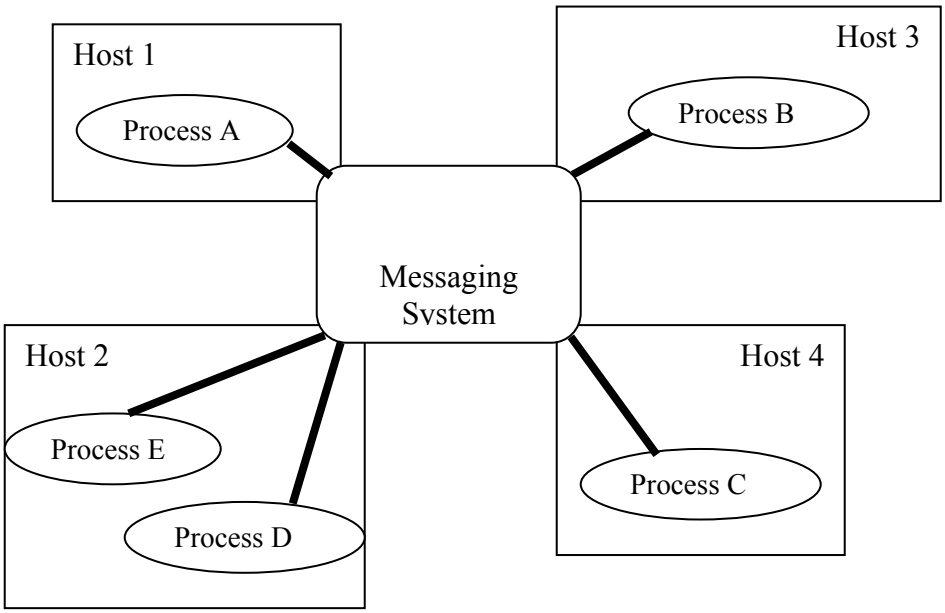


Figure 1.2. Virtually fully connected networks.

1.2 Requirements for Alert Server

If the Alert Server provided a union of all the existing features of messaging systems it would be much too complicated for its intended users and would suffer from considerable unnecessary overhead. It is crucial that Alert Server includes the functionality needed to implement sophisticated enterprise applications. The alert server attempts to minimize the set of concepts a programmer must learn to use enterprise-messaging products. Alert Server strives to maximize portability and also assures priority-based delivery of alerts. Some of the requirements of Alert Server are:

1. Alert Server should implement a publish/subscribe model. In publish/subscribe, programs subscribe (register interest in) a subject. Programs also publish (send) messages to the subject. Once a program has subscribed to a subject, the program will receive any messages published to that subject in the distributed application. The application developer defines subjects. This approach is most appropriate for highly distributed applications where fault tolerance and high performance is needed.

2. Alert Server should deliver alerts before they expire (time-to-live) based on priority. The Alert Server should guarantee the delivery of alerts on time. This is an important requirement since a client may not be interested in some of the alerts after sometime.
3. Dynamic delivery of alerts between multiple producers and consumers based on their registration/subscription topics. The delivery and publishing of alerts should be asynchronous. Clients should not block while waiting for alerts.
4. Alerts (messages) and acknowledges should be persisted to handle crash recovery of server. Alerts should be persisted when specified by user. Alert Server should be able to recover from crashes. It should have the ability to run in normal and crash recovery mode.
5. Privacy and integrity of alerts should be maintained while passing between client applications. Privacy of messages based on the relative importance of alerts.
6. C, C++ and JAVA clients should be handled equally. Alerts should be distributed irrespective of the programming language of the client applications.
7. Alert distribution should be possible in a distributed environment.
8. Dynamic monitoring and administration of (multiple) alert servers should be possible.
9. Alert Server should be integrated with LED/GED [1, 3] so that clients can publish/subscribe for alerts and produce/consume events.
10. Consumers should be able to subscribe to multiple topics in one request.

1.3 Requirements of Alert Clients

Alert client can be either a producer or a consumer or both. The alert producer does not necessarily need to know who the receiver(s) of the message will be. The producer "publishes/sends" the messages to the Alert Server that is responsible for the distribution of messages. Alert Consumers are responsible for processing and responding (or taking other appropriate actions) to the alert (message) or a set of alerts by

subscribing/registering through the alert server. Alert Server provides APIs to implement the clients needed. An important goal in this case is to minimize the work needed to implement a producer or a consumer.

1.4 Approaches for message distribution

1.4.1 Client/Server Architecture with RPC

In this architecture, different applications communicate by making remote procedure calls in which the messages are sent. Remote Procedure Calls (RPC) [4] are embedded within client applications. Because they are embedded, RPCs do not stand alone as a discreet middleware layer. When the client program is compiled, the compiler creates a local stub for the client portion and another stub for the server portion of the application. These stubs are invoked when the application requires a remote function and typically support synchronous calls between clients and servers. Thus this architecture reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces [5]. However, RPC is appropriate for client/server applications in which the client can issue a request and wait for the server's response before continuing its own processing. Because most RPC implementations do not support peer-to-peer, or asynchronous client/server interaction, RPC is not well suited for applications involving distributed objects or object-oriented programming. Asynchronous and synchronous mechanisms each have strengths and weaknesses that should be considered when designing any specific application. In contrast to asynchronous mechanisms employed by Message-Oriented Middleware, the use of a synchronous request-reply mechanism in RPC requires that the client and server are always available and functioning (i.e., the client or server is not blocked). In order to allow a client/server application to recover from a blocked condition, an implementation

of a RPC is required to provide mechanisms such as error messages, request timers, retransmissions, or redirection to an alternate server. The complexity of the application using a RPC is dependent on the sophistication of the specific RPC implementation (i.e., the more sophisticated the recovery mechanisms supported by RPC, the less complex the application utilizing the RPC is required to be). RPCs that implement asynchronous mechanisms are very few and are difficult (complex) to implement. When utilizing RPC over a distributed network, the performance (or load) of the network should be considered. One of the strengths of RPC is that the synchronous, blocking mechanism of RPC guards against overloading a network, unlike the asynchronous mechanism of Message-Oriented Middleware (MOM) [6]. However, when recovery mechanisms, such as retransmissions, are employed by an RPC application, the resulting load on a network may increase, making the application inappropriate for a congested network. Also, because RPC uses static routing tables established at compile-time, the ability to perform load balancing across a network is difficult and should be considered when designing an RPC-based application.

1.4.2 Client/Server Architecture with ORB

An object request broker (ORB) is a middleware technology that manages communication and data exchange between objects. ORBs promote interoperability of distributed object systems because they enable users to build systems by piecing together objects—from different vendors—that communicate with each other via the ORB. The implementation details of the ORB [7] are generally not important to developers building distributed systems. The developers are only concerned with the object interface details. This form of information hiding enhances system maintainability since the object communication details are hidden from the developers and isolated in the ORB. The two major ORB technologies are Object Management Group's (OMG) Common Object

Request Broker Architecture specification and Microsoft's Common Object Model. Even though CORBA [8] has certain advantages, it does not support the transfer of objects. There is also no garbage collection. Moreover ORBs developed by different vendors may have significantly different features and capabilities. Thus, users must learn a specification; the way vendors implement the specification, and their value-added features (which are often necessary to make a CORBA product usable). Similarly COM/DCOM [9, 10] has certain negative aspects that make it unsuitable in implementing such systems. It is platform specific and there is no stability of APIs making maintainability of software difficult in the long run.

1.4.3 Client/Server Architecture with MOM

Message-oriented middleware (MOM) is a client/server infrastructure that increases the interoperability, portability, and flexibility of an application by allowing the application to be distributed over multiple heterogeneous platforms. It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces [5]. MOMs generally support synchronous calls between clients and server applications. Message queues provide temporary storage when the destination program is busy or not connected. MOM reduces the involvement of application developers with the complexity of the master-slave nature of the client/server mechanism. MOM increases the flexibility of architecture by enabling applications to exchange messages with other programs without having to know what platform or processor the other application resides on within the network. The aforementioned messages can contain formatted data, requests for action, or both. Nominally, MOM systems provide a message queue between interoperating processes, so if the destination process is busy, the message is held in a temporary storage location until it can be

processed. MOM is typically asynchronous and peer-to-peer, but most implementations support synchronous message passing as well. MOM is typically implemented as a proprietary product, which means MOM implementations are nominally incompatible with other MOM implementations. Using a single implementation of a MOM in a system will most likely result in a dependence on that MOM vendor for maintenance support and future enhancements. This could have a highly negative impact on a system's flexibility, maintainability, portability, and interoperability. It is for these reasons that the Alert Server is built using the queuing mechanism that MOMs use and Java RMI for the transfer of messages between different distributing processes.

1.5 Need for a messaging system

Message passing is not as sophisticated and robust as distributed objects, and is relatively simple to implement. So why do we need a message passing system? The goals of the approaches explained earlier are very different. Distributing objects (RMI and CORBA) extends an application across the network by making its objects appear to be spread across the hosts in our virtual machine. Message passing serves a much simpler role, defining a simple communication protocol for sending data/objects. Passing messages avoids the communication overhead involved in using most distributed object schemes, and does not require any special network protocols. So message passing is useful and sometimes a necessary tool, particularly in the following situations:

1. Communication needs are relatively simple in nature.
2. Transaction throughput is critical.
3. The scope of the system is limited, so that rapid implementation takes precedence over sophistication and flexibility of design.
4. Special network protocols need to be avoided (e.g., parts of the system need to operate behind a firewall).

1.6 Existing Messaging Systems

Messaging systems are peer-to-peer facilities. In general, each client can send messages to, and receive messages from any client. Each client connects to a messaging agent, which provides facilities for creating, sending and receiving messages. Each system provides a way of addressing messages. Each provides a way to create a message and fill it with data. Some systems are capable of broadcasting a message to many destinations. Others only support sending a message to a single destination. Some systems provide facilities for asynchronous propagation of messages (messages are delivered to a client as they arrive). Others support only synchronous propagation (a client must request each message). Each messaging system typically provides a range of service that can be selected on a per message basis. One important attribute is the lengths to which the system will go to insure delivery. This varies from simple best effort, only once delivery, and guaranteed delivery. Other important attributes are message time-to-live, priority and whether a response is required. A few of these messaging systems are discussed in chapter 2.

Alert Server uses both the distributed objects approach and the MOM approach. It has to deliver alerts and should work in a distributed environment. Therefore we use a combination of both the approaches to make communication look simple and provide a little bit of sophistication and flexibility. This thesis discusses the design and implementation of Alert Server. Alert Server is the alert and acknowledgement message queue and distribution mechanism. It is a persistent and recoverable middleware approach to alert distribution. It maintains transaction logs for a comprehensive audit trail of alerts, acknowledgements and receipts to appropriate destination. In general, an alert is generated and submitted to the Alerts Server based upon some mission application criteria or condition. At the alerts server, the alerts are logged and queued and if necessary persisted. The Alerts Server determines if there are any subscribers for this

alert and if so, forwards it to the destination. An alert producer could be a human operator who “fills in the blanks” of an alert message through a GUI or other means. Alert producers can also be software components that execute “under the hood,” invisible to human operators. An alert producer assembles the alert in reaction to some system condition and the send it to the distribution process, Alert Server. Alert consumers are those applications that are interested in receiving (a subset of) alerts. This is always accomplished via “registering” or “subscribing” for alerts that contain a particular pattern in the alert destination or topic data element by specifying a filter during alert registration. Any client application can be either an alert producer or a consumer or both as long as they use the Alert Server APIs to talk to the Alert Server. A proxy server has also been designed to handle C/C++ clients in a similar way as JAVA clients.

CHAPTER 2

OVERVIEW OF RELATED WORK

With all the power that the messaging systems have to offer it is not surprising that there are many such systems available in the market, each with one with its own set of advantages and disadvantages. This chapter discusses some of these systems that are available in the context of meeting our requirements.

2.1 Java Message Queue

iPlanet's Java Message Queue (JMQ) [11] product is a standard based solution to the problem of inter-application communication and reliable data transmission across networks. With JMQ, processes running in different architectures and operating systems can simply connect to the same virtual network to send and receive information. This uses a publish/subscribe model. Based on Java Message Service (JMS) [12] open standard, JMQ applications are designed to be easily portable across different computer architectures and operating systems and to handle all data translation between application processes. Even though, JMQ provides delivery of messages on the basis of priority and supports persistence, it cannot be used to transfer messages between different client applications (LED/GED) [1, 3] because it does not have any crash recovery mechanism. Apart from this, the actual length of time the messages are held in the queue and the consequences of resource overflow are not handled. This needs to be addressed in a message-distributing environment, as one cannot assume infinite resources, as a lot of messages need to be handled at any point in time. More over, the burden of initialization of the queues and assigning these queues to specific topics (subjects) so that messages are stored on the system falls on the administrator. There is also

no provision for clients to set some of the attributes of the message like persistence, priority etc. These attributes are set by send/receive methods of the Java Message Queue instead.

2.2 IBM MQ Series

MQ Series [13] from IBM is one of the most established messaging products in the industry. MQ Series has traditionally been oriented towards queue-based messaging, but enhanced in late 1999 to support publish/subscribe messaging and a JMS interface. The messaging clients communicate with one or more queue managers, which are implemented in native code and are available for a wide range of platforms. A queue manager is a program that provides messaging services to applications. Applications that use interface called Message Queue Interface (MQI) to put and get messages from queues. The queue manager ensures that messages are routed to another queue manager. The communication between them is through channels. There are two types of channels. A message channel is a unidirectional communication link between two queue managers that is used to transfer messages between them. An MQI channel is bi-directional and connects an application (MQI client) to a queue manager on a server machine for the transfer of MQI calls and responses. Queue managers are grouped together into clusters. A cluster [14] is a group of two or more queue managers that are logically associated and can share information with each other. Any queue manager can send a message to any other queue manager in the same cluster without the need for user to set up a specific channel definition because all this information is held in a repository to which all queue managers in the cluster have access. This creates a problem if the computer on which the repository uses DHCP (dynamic allocation of IP Address) since MQ series uses IP address to find repository. Hence, if the address changes, other queue managers will no longer be able to find the repository. This still applies even if all queue managers in the cluster will be on the same computer, because the IP address is still used to find the repository. Thus, MQ series cannot handle changes to network easily as it is pre-

configured to forward messages from a queue to another queue. If any component fails in this process, the message queuing software is stuck until the failed component is functioning correctly as it does not support recovery mechanism. More over, it requires at least one routing process on every machine. This is not reasonable when deploying to many machines. This software also requires special privileges to install or run, as changes must be made to the operating system kernel on most supported platforms. In addition to these, MQ Series does not support subscription to messages on the basis of regular expressions and crash recovery.

2.3 Sonic MQ

SonicMQ [15], from Sonic Software, provides a hub-spoke implementation of JMS pub/sub ass well as point-point domains written entirely in JAVA. SonicMQ also provides for JMS extensions like XML messages and server clustering. The ability of a messaging server to deliver messages at a constant rage (regardless of publisher speeds and the number of connections to the server) depends to a large extent on flow control algorithms employed by them. In a typical messaging environment, message producers are usually faster than the consumers of the message. To ensure the capacity limits (memory, threads, etc.) within server, the sending client must be throttled to prevent the loss of messages or a buffer manager must be provided on the server. A particularly important aspect of any flow algorithm in a publish/subscribe messaging server is to ensure that if a particular subscriber slows down, other subscribers on the same topic are not adversely affected. That is, a single slow subscriber should not slow down the entire system. SonicMQ handles this problem by buffering messages internally and throttles publishers when internal buffers overflow. This actually slows down all publishers to the speed of the slowest consumer. This leads to serious issues while using this product in real world application as the publishers are blocked till subscribers catch up. Even though, SonicMQ claims to support distributed transactions, it

might fail under some circumstances: if a publish call is part of a distributed transaction and the call blocks forever, then the Distributed Transaction Coordinator) would either time-out this transaction or report a state of ambiguity. More over, SonicMQ uses pre-fetching by default, resulting in misleading performance results sometimes. Some applications that are not aware of pre-fetching under the covers can run into potentially serious issues at the deployment stage, when some messages might not be delivered to any other receiver because they are queued in a single receiver (which might have crashed/hung due to potential application failure). This results in messages not getting processed even while some queue receivers are waiting for messages all the time. SonicMQ server resources (threads) are consumed just waiting for incoming data, adding unnecessary overhead. Furthermore, as each additional client connects a new thread needs to be allocated on the server, leading to linearly increasing server load. Under such conditions the server eventually slows down to unacceptable levels or crashes. In typical cases, allocating more than 2000 threads is impractical on a single JVM, although the precise limits vary depending on the hardware and operating systems platforms used. SonicMQ uses Cloudscape, which is a Java based database management system instead of a file-based data store to persist and store messages. This reduces the system speed as file-based data store delivers the messages 10-15 times faster than delivering messages from database.

2.4 Global Event Detector

The Global Event Detector (GED) [1], developed at ITLAB at UT Arlington, is a server based on the notification/subscription paradigm. All messages passing is done in a demand-driven mode. That is, no messages are sent to the server unless there is a consumer for that message. The server receives an event detection request from a consumer application and forwards it to the corresponding producer only when it registers with the server. When the event of interest defined in the producer application occurs, the producer application

notifies the occurrence of event to the server. The server not only forwards the occurrence of the event to the corresponding consumers, but it is also responsible for detecting any composite event based on that event. GED uses the ECA (Event-Condition-Action) rule paradigm in order to support active capability in a distributed environment. According to the ECA rule paradigm, whenever the event occurs, the condition defined in the rule (for that event) is evaluated and the corresponding action is performed if applicable. GED does not guarantee delivery of events and implements notification/subscription paradigm. This fine in case of event detection but if the consumer is just interested in the generation of event and not the event detection then a publish/subscribe paradigm would be more better. It also does not provide any filtering mechanism so that the server sends only the events that are of interest to the consumer. There is also no mechanism that supports priority based delivery and a way to specify that the event should remain on the server for a certain duration i.e., the time to live header. GED can be used in an environment where in client applications need to detect events. But if the applications are just interested that the event has occurred then this server may not be that suitable.

2.5 Common Object Request Broker Architecture

Based on publish/subscribe paradigm, event service is one of the services from Common Object Request Broker Architecture (CORBA) [8]. It defines three roles (supplier, consumer, and event channel) [16]. The suppliers and consumers are decoupled, and transparent from each other. Suppliers can push data to consumers through the event channel. Likewise, consumers can use event channel to pull the data from suppliers. The event channel works as a mediator between consumers and suppliers. The event channel interface can be used for adding consumers, adding suppliers, and also for destroying the channel.

There are four models of component collaboration in the event service architecture: Push Model, Pull Model, Hybrid Push/Pull Model, and Hybrid Pull/Push Model. The Push

model allows suppliers to initiate the transfer of event data to consumers. In the pull model, the consumers request the event data from suppliers through event channel. The Hybrid Push/Pull model allows both consumers and producers to initiate the transfer of event data. The event channel plays the role of a passive mediator. The active consumers can request data via the event channel in which the active supplier pushes the data. The Hybrid Pull/Push model, in contrast to the Hybrid Push/Pull model, allows the active event channel to pull the data from suppliers and push them to consumers.

Although the symmetry provided by the COBRA event service is well designed, it is not possible for consumers to subscribe only to events, which are of interest. Each event that is sent from each producer to the event channel is delivered to all the registered consumers. This requires consumers to filter out event data that is not of interest. It involves additional overhead for the consumers. This is likely to increase the network utilization and cause clogged network traffic. In addition, there is no notion of priority based delivery and regular expression based subscription for events. More there is also no notion of crash recovery and the idea of persisting the events.

CHAPTER 3

DESIGN OF ALERT SERVER

Message-oriented middleware (MOM) refers to data transmission between front-end applications and back-end applications. Middleware can be composed of many layers, each addressing the specific requirements of inter-application communication. As indicated, the purpose of a messaging system is to accept messages (data/objects) from one client for delivery to another. Typically, the system is designed and implemented in a manner that promises some degree of performance, reliability, availability and security. A messaging system is composed of client applications, messages, and the messaging service. A client application refers to one or more processes that coordinate to implement some functionality. These processes may be located on a single machine or distributed across different machines in different locations. A client application might itself be a part of a large application. The messages contain not only data but information about the data as well. Messages are sent in a pre-defined format understood by the sending client process and the receiving client process. The message service includes:

1. A message server (sometimes called “message router” or a “message broker”). Alert Server acts as a message broker for our messaging system.
2. Administration tools to manage the message server and messages in the system.
3. An API for creating messages and interacting with the service.

This chapter discusses the design of Alert Server. First, it outlines the functionality to be supported by the Alert Server followed by design choices to achieve

this functionality. It also discusses the contents of an Alert needed for efficient and easy way of passing messages.

3.1 Functionality to be supported

The Alert Server should provide APIs to send messages from one application to another. The client applications should also be able to register and unregister topics of interest. They should be able to cancel or revoke messages (that have been sent) and should also be able to send acknowledgements and receipts. In addition, Alert Server should have delivery logic in order to send, and if necessary, persist, messages on the basis of their priority. Finally, the alert server should be able to recover from crashes to ensure no loss of alerts and proper continued delivery of alerts. The server should also handle acknowledgements and receipts apart from alerts to guarantee delivery of alerts. But before we get into the design of Alert Server, it is important to understand what alert, acknowledgement and receipt is and what constitutes them.

3.2 Alert

An alert message contains an alert header as well as an optional alert body. All messages support the same set of header fields. Header fields contain values used by both clients and the Alert Server to identify and route messages. Body, on the other hand can be any Java Object for Java-based clients and 1024 characters in length for C/C++ clients. A header of the alert message has the following data elements.

Destination (Topic): The destination field in the data element is the “topic” and synonymous to, for example, a message “topic” or an email “subject”. The destination data element provides the information needed by the Alert Server to distribute the alert to proper recipients. It is the value in this field the alert consumers register or unregister by specifying a pattern in the registration request. There are three different types of topic that can be provided: TAG, PROFILE and USER. Each of these topic types are specified

using the form “topic type: value[, value]”. Topic type is a reserved word that must be one of a “TAG”, “PROFILE”, or “USER”. The value of this field must begin with one of 3 prefixes (with colon included and all the letters capitalized):

TAG:

PROFILE:

USER:

Every alert contains the destination or a topic header field. The destination data element provides the information needed by the Alert Server to distribute this alert to the proper recipients.

TAG: The “TAG:” prefix helps alert consumers to subscribe to receive specific alerts by specifying a filtering mechanism that employs UNIX regular expression masks. For example: TAG:.* specifies all alerts. TAG:ABC specifies all alerts where destination contains the string ABC. TAG:^A.* specifies all alerts where destination begins with A. TAG:.*X$ specifies all alerts where destination ends with X.

PROFILE: This prefix helps alert consumers to subscribe to any message that belong to a specific profile. For example: PROFILE:Watchman, PROFILE:Student.

USER: The USER: prefix as the name indicates helps in subscribing to messages from a particular user name. For example: USER:A, USER:B, USER:C.

When the alert server processes an incoming alert, it places the filtering mask that the consumers subscribed to the pattern contained in the alert’s destination. For example, if there is a consumer subscribed to a filter of “TAG:Alert” and an alert with the destination “TAG:AlertXYZ” comes in, then the subscription “TAG:Alert” is matched against the topic or destination “TAG:AlertXYZ”. The regular expression mask of “TAG:Alert” placed against the topic “TAG:AlertXYZ” matches true (“TAG:Alert is a sub string of “TAG:AlertXYZ”). In the case of the other prefixes, the matching is performed by comparing the strings in the subscription filter with the destination string in

the alert header. Consumer clients have the option to subscribe to multiple topics using separators. Multiple topics with the same prefix are submitted by separating with commas, for example, TAG:a, b, c will produce three subscriptions; TAG:a, TAG:b, and TAG:c. Similarly, multiple topics with multiple prefixes are submitted using a semicolon to separate prefixes, for example, TAG:a, b;USER:a, b will produce four subscriptions, TAG:a, TAG:b, USER:a, USER:b.

Alert Type: This field in the header identifies the message type. There are basically two types of alerts, alert itself and an acknowledgement for the alert. An alert is specified by setting the alert type data element value to Alert, where as an acknowledgement is specified by setting the alert type to Acknowledgement.

Duration: Duration data element in the header helps in identifying the messages that have expired. Alerts or acknowledgements will remain active and available for distribution from the Alert Server according to its time-to-live indicator. Once the alert server receives and forwards the message to any registered recipients, it will remain in the Server's queue for the specific duration. The Server deletes the expired messages when it tries to distribute it to the clients. Indefinite storage of messages in the queues is handled by setting this data field to zero. These alerts should be explicitly cancelled by the original producer or by any client as specified by a cancel policy.

Priority: This field in the header helps in priority based delivery of messages. The priority levels are 1-10, where 1 is designated lowest priority and 10 is designated highest priority. Higher priority alerts need to be distributed before the lower priority alerts and messages having the same priority are delivered in the order they arrive i.e., on a first come first serve basis.

Classification: This part of header information allows application specific classification. Unclassified, Confidential, Secret and Top Secret are the four classification levels.

Classification of an alert basically defines the privacy level of alert. This data field is reserved for future use.

Persistence: The producer designates messages as persistent by setting this field in the header. The Alerts Server stores persistent messages so that they are reloaded in the case of restart and recovered in the case of crash. The storing of the messages and their retrieval is discussed later.

Acknowledge Policy: This field ensures the delivery of messages to the destination. An alert can have one of the three acknowledgement policies attached to it: None, Client Acknowledgement, and Client Receipt. A client acknowledgement requires the receiving client to generate an acknowledgement alert where as a client receipt is automatically constructed and submitted to the server after the client is notified of an incoming alert. Client receipts are not stored on the server as clients make blocking calls when they send alerts that are receipts. Also alert server discards acknowledgement for alerts that do not require acknowledgement. The generation of alert acknowledgements and receipts are explained in the implementation chapter of alert server.

Cancel Policy: This field helps in the cancellation of alerts that live indefinitely on the server. An alert can be cancelled i.e., deleted from the queue on the server, by any client if the Cancel Policy field is set to ANY, or only by the producer of the alert if it is set to the ORIGINATOR.

Scope: This data element is reserved for future use. This has been included in order to decide the scope of the message when there are a many alert servers. This is set to zero if the scope has to be limited to one alert server.

Alert ID: Alert ID is a unique integer that is generated by the alert server to identify a particular alert or acknowledgment.

Correlation ID: This data element is only used when the message is an acknowledgment. This field takes on the value of the Alert ID for which it is an acknowledgement.

Alert Body: The alert body is an object that can be sent with the message in the case of Java alert producers while it is a character string in the case of C/C++ alert producers. This difference is due to the limitation of C and C++. The body in the case of acknowledgement is the string "ACK."

Alert Hash Table: This is just a hash table that holds the mapping between the different consumer lists in the consumer table data structure and the last consumer that has received this alert. Only server for distribution purposes, accesses this alert attribute. Clients cannot access this attribute. The attribute is absent when the alert is generated on the client; it is added by the Alert Server when the alert is distributed to the consumers. The usage and contents of this hash table are explained in the implementation chapter 4.

An example of alert with its header and body is shown figure 3.1:

ALERT HEADER:		
Alert ID	: 1	[system generated]
Destination	:"TAG: .*X"	[TAG, USER, PROFILE]
Arrival Time	: 92115678	[Time received on server]
Duration	: 10000	[Time to live on server]
Type	: ALERT	[ALERT, ACK]
Persistence	: TRUE	
Scope	: 0	[Specific to this server]
Priority	: 8	
Classification	: "UNCLASSIFIED"	
Cancel policy	: ORIGINATOR	[ORIGINATOR, ANY]
Acknowledgement policy	: NONE	[NONE, CLIENTACK, RCPT]
User ID	: "TRLP1"	
Correlation ID	: 0	[Set only for acknowledgement]
ALERT BODY:		
Body = "this is the optional part of alert message"		[Can be an object]

Figure 3.1. An Example of alert format.

3.3 Acknowledgement

An alert acknowledgement as explained above is actually an alert. It contains an alert header with the same destination or topic as its associated alert, origination time, persistence, time-to-live (duration) etc. except for the correlation ID. The correlation ID data element contains the alert ID value of the alert to which this acknowledgment message is responding. The body for an acknowledgement contains the string "ACK". Alert consumer applications may register just for alert acknowledgements, without having to register for an alert. This capability is provided so that application developers can create a chain of events. For example: consider A to produce an alert, B to register for this alert, C to register for acknowledgements for this alert. A sends alert, B receives alert and acknowledges it, C receives acknowledgement (and then goes off to do something else, for example).

3.4 Receipt

An alert producer can request a "delivery" receipt for an alert by setting alert's acknowledgement policy data field to Client Receipt. The Alert Server constructs the receipt automatically the instant the alert consumer receives the alert and then forwards it to the requesting producer. Unlike acknowledgements, receipts are not queued or stored at the Server, nor can alert consumers register for receipts. This feature is simply to ensure the delivery of the alert.

3.5 Alert Server Architecture

This section explains the different design choices available and the reason for choosing one of them.

3.5.1 Messaging Models

Most messaging products support either point-to-point or the publish/subscribe approach to messaging.

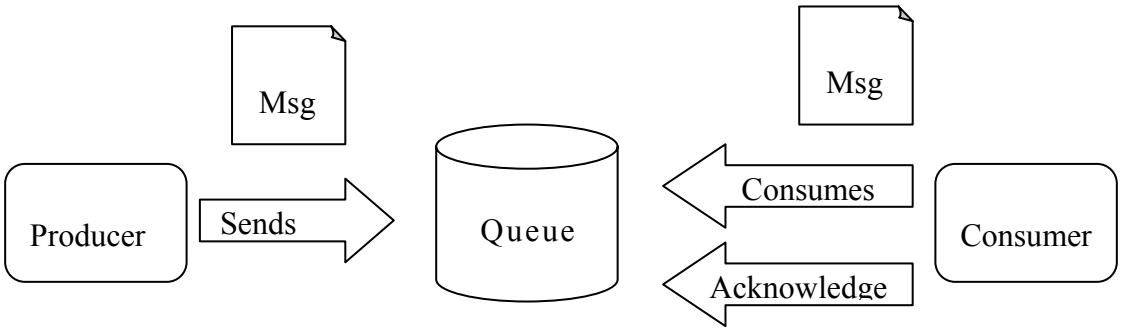


Figure 3.2. Point-to-point messaging model.

Point-to-point messaging model: A point-to-point (PTP) product or application is built around the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queue(s) established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire. PTP messaging has the following characteristics:

1. Each message has only one consumer.
2. There are no timing dependencies between a sender and a receiver of a message. The receiver can fetch the message whether or not it was running when the client sent the message.
3. The receiver acknowledges the successful processing of a message.

This messaging model is shown in figure 3.2. This messaging model is useful when the message needs to be processed successfully by one consumer. But Alert Server has to handle multiple clients simultaneously, therefore publish/subscribe messaging model has been chosen over this messaging model.

Publish/Subscribe Messaging model: In a publish/subscribe (pub/sub) model, clients address messages to a topic (destination). Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Pub/sub messaging has the following characteristics:

1. Each message may have multiple consumers.
2. There is a timing dependency between publishers and subscribers, because a client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

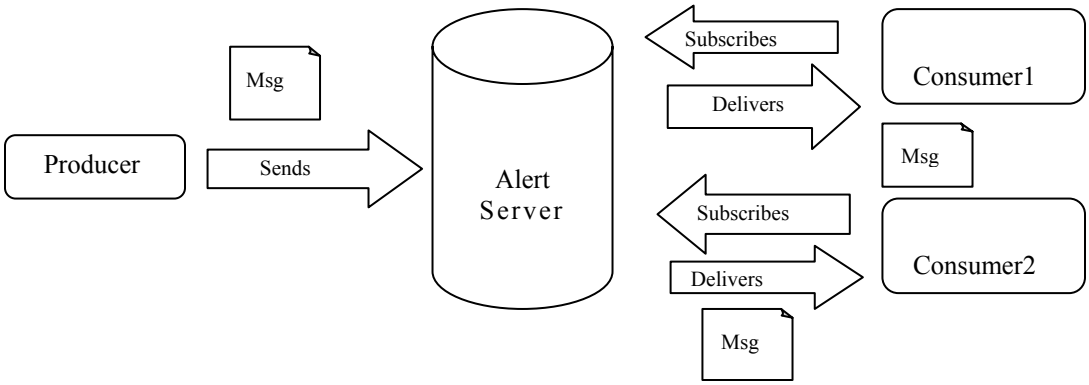


Figure 3.3. Publish/Subscribe messaging model.

The Alert Server uses this messaging model as it has to delivery messages to zero, one or many consumers that are anonymous. This timing dependency is relaxed by allowing the producers to create persistent alerts. Persistent alerts can be received even when the subscribers are not active. Thus, persistent alerts provide the durability and reliability provided by the queues and still allow clients to send alerts to many consumers. A sending client addresses (publishes) the message to a *topic* to which multiple clients subscribe. There can be multiple publishers, as well as subscribers, to a topic. In a publish-and-subscribe system, a client can be a publisher (message producer), a subscriber (message consumer), or both. The delivery of messages to multiple clients, as well as future subscribers, makes this model a choice on which the Alert Server architecture is based. This messaging model is shown in figure 3.3.

3.5.2 Message Consumption

Architecture of a server also decides the way in which the clients consume messages. Messages can be consumed in either of the two ways.

1. Synchronously: A subscriber or a receiver explicitly fetches the message from the destination by calling a method. The method can block until a message arrives, or it can time out if a message does not arrive within a specified time limit.
2. Asynchronously: A subscriber need not wait for the delivery of the message. Whenever the message arrives, the server forwards it to the consumers that have registered for that message. The consumers do not have to wait for the delivery of the message. Alert Server uses this mode of message consumption, as it is essential that the clients do not block during the publishing of alerts.

3.5.3 Message Delivery Mode

Architecture of Alert Server should guarantee the delivery of messages. For this it is essential to persist the messages or alerts. But this is cannot achieved without overhead. Therefore Alert Server architecture supports two modes of message delivery and leaves it upon the client application to decide the delivery mode of messages. Clients can do this by setting the delivery mode in the persistent header field. More over messages also need to be persisted so that the architecture can support crash recovery.

1. The NON-PERSISTENT mode is the lowest-overhead delivery mode because it does not require that the message be logged to stable storage. Alert Server failure can cause a NON-PERSISTENT message to be lost.
2. The PERSISTENT mode instructs the Alert Server to take extra care to insure that the message is not lost in transit due to its failure. This mode requires server to log the alerts and retrieve them during normal startups as well as in the case of recovery after crashing. A message that is persisted can expire. These messages are removed from the stable storage when the server reloads. Merely persisting the alerts (i.e., writing

the alerts into a stable storage) does not help. There should be a mechanism for fast retrieval of alerts from the logs. This mechanism is explained next.

Logging and Retrieval Mechanism

1. BLSN (4 bytes)
2. DLSN (8 bytes)
3. Index Table (12 bytes)

LSN	fp	Size	Cancel bit
1	20	24	1
3	45	27	0

Alert with ID 1
Alert with ID 3
.
.
.
.

Figure 3.4. Contents of log file.

The persistent mode delivery of the alerts entails them to be stored to a stable storage (i.e., the disk in our case). Therefore the logging and retrieval of alerts should be done efficiently. A naïve approach is to write the alerts into a log and scan the log sequentially for retrieving. But this involves a lot of search time. Another approach is to store the alerts in a database. But this involves more overhead than storing a file. For this reason, a separate logging and retrieval mechanism has been designed. The alerts are stored in log files depending on their priority. For example: all alerts with priority level 1 are stored in one file and all alerts with priority level 2 are stored in another separate file. There is a file for each priority level. The alerts are serialized and then written into the logs. The main reason for doing this is to reduce the I/O time as serialization helps in easy reading and writing of entire objects and primitive data types, without converting to/from raw bytes or parsing clumsy text data. But just writing these serialized objects

would not help in finding an alert in log with hundreds of them. Therefore for fast retrieval, some other information is also stored in the log along with the alerts. All the contents of the log are serialized for the above reason. The contents of the log file are shown in figure 3.4 and are described in detail below:

Index Table:

Index table is a data structure that is stored in the log file for searching and retrieving alerts. Each log file has an index table to store and retrieve the alerts belonging to its priority. The index table is also stored in a serialized manner. The table has records that hold the information of the location of the alert in the log. Each table record has a Log Sequence Number (LSN) to identify the alert (it is actually alert ID), pointer to the position of storage of the alert in the file (fp) to index to that position in the file where alert is stored, size of the alert (size) to read the bytes pertaining to that alert and a cancel bit to help in identifying the cancelled alerts. LSN of a record helps in indexing into the table and finding the record that belongs to alert of interest. Then following the file pointer (fp) in that record and reading the number of bytes as specified by the size field (size) in the index table record, alert is read back from the log. The storing of the alert is done in a similar way. New table record is always created and inserted into the corresponding index table in the next available slot. The records are added sequentially. Since the index table holds records that indicate the position of alerts, this also needs to be stored along with alerts. Thus, any alert retrieval requires first reading the index table in the log.

Buffered Log Sequence Number (BLSN):

BLSN in each file is an integer stored in each log along with the index table. BLSN is set to the ID of the alert that has recently been added to the priority queue. There is a BLSN for each priority queue in its corresponding log file. This integer helps in reducing the

time for building the priority queue back when the server recovers from crash. Only those alerts that are between BLSN and DLSN are put back into the priority queue. As all the other alerts that are not between BLSN and DLSN have already been sent.

Delivered Log Sequence Number (DLSN):

DLSN, like BLSN, is also an integer stored in each log for corresponding priority queue. DLSN, unlike BLSN, is set to the ID of the last alert that has been sent from the priority queue and received by the client.

Cancel bit:

The cancel bit indicates whether the alert has been cancelled or not. This bit is set when the alert is cancelled. Only those alerts that are not cancelled are read back when the server starts up or when the server recovers from a crash.

Both DLSN and BLSN help in the retrieval of NON-PERSISTENT alerts in the case of crash recovery of the Alert Server. Instead of reading *all* the non-persistent alerts from the log when the server recovers only those alerts whose IDs fall between BLSN and DLSN are read from the log and put into the priority queue. The serialized log files always contain BLSN at the start of the log, followed by DLSN is at 4 bytes followed by index table with its records at 12 bytes. All the alerts are stored after index table. The alerts after being stored in the log are sent to the queue for distribution to the consumers who have subscribed to their topics. Since acknowledgements are alerts with the correlation ID set to that of their alerts, they are handled as if they are alerts. The handling of the subscription of the alerts is described in the next section.

3.6 Subscription and Unsubscription of alerts and acknowledgement

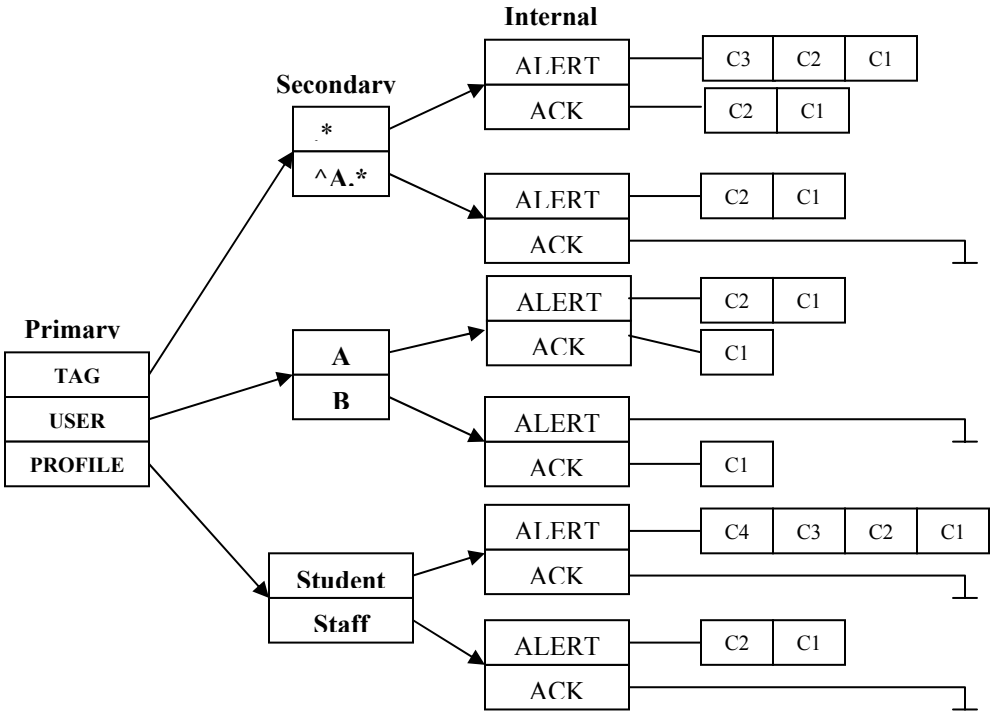


Figure 3.5. Consumer table data structures.

Consumers that have subscribed for alerts or acknowledgements need to be stored so that the server can handle their requests. Therefore the subscription and unsubscription of consumers has to satisfy a number of requirements:

- Consumers should be able to subscribe/unsubscribe to an alert dynamically.
- Consumers should receive alerts as they arrive i.e., they have to receive alerts on a first come first serve basis.
- Since alert delivery is based on priority, it is essential that the consumers for alerts be found efficiently.
- Consumers should not receive the same alerts.

Since a number of consumers register with the server at any instant of time, it is essential that the server use a data structure that helps in categorizing the clients upon the

alerts they subscribe and thus help in the efficient distribution of alerts. For this purpose, an extended hash table has been used. This data structure is shown in figure 3.5.

This data structure contains a primary hash tables with three buckets. Primary hash table helps in categorizing the subscriptions on the basis of topic types. Three buckets refer to the three topic types that are supported by the alert server, namely, TAG, USER and PROFILE. Each bucket points to another hash table. This table at the second level is called the secondary hash table. This table categorizes the consumer subscription pertaining to different subjects of the same topic type. As the subscription of consumers with different subject increases, the number of buckets in the secondary hash table also increases. Each bucket in the secondary table points to an internal hash table. This internal hash table has alert type (data field that decides whether the message is an alert or an acknowledgement) as its key. A hash table has been chosen instead of array so that in future subscription may be extended to receipts. There are always two buckets for each internal hash table, one for alerts and one for acknowledgements. These buckets have consumer lists as their keys. Each list in the entire data structure has unique ID called the List ID. List contains consumer nodes that hold information regarding different consumers registered with the Alert Server. Each consumer node in a list has an ID attribute that indicates its arrival in the list. Apart from consumer nodes, the lists also contain information regarding the number of consumers added and number of consumers deleted. For each consumer node added or deleted to the list these attribute of the list change accordingly. These attributes help in reducing the searching for the consumer that needs to receive alert within the list. It also prevents the resending of alerts to the same consumer again. This is explained in detail in the chapter 4.

Thus categorizing the alerts helps in finding the consumers of the alert efficiently. Consumer registration causes a consumer node to be added to the beginning of its respective list. Hash tables have been chosen (over other possibilities, such as

linked lists and arrays) in all the cases as they reduce search time. A hash table also allows one to add new keys easily without much change in code if needed in the future. The unsubscription for alerts and acknowledgements by a consumer is done in a similar way. Unsubscription results in the deletion of that consumer node from the consumer list preventing the Alert Server from distributing alerts to consumers that have unsubscribed. The unsubscribed consumer node is found by indexing through the hash table and traversing through the list. Maintaining the consumer information in the lists in the hash table help in improving concurrency. Apart from the consumer nodes, the list has a LIST ID and LIST ADDED attributes that help in traversing the entire list and also preventing the resending of the same alerts to the same consumer. The queuing and distribution of alerts is explained later.

3.7 Queuing and distribution of alerts

The alerts are stored in the logs depending on their persistent mode and later put into the priority queue. There are ten queues, one for each priority level 1 through 10. The data structure used for the queues is an array of queues. The size of this array is ten corresponding to the number of priority levels. Alerts are stored in the queue on the basis of their priority. So all alerts with the equal priority are stored in the same queue. Every new alert is always added at the beginning of its queue in order to preserve its semantics. The priority data field in the alert header is used for indexing into the array and getting the queue at that index. Therefore insertion of the alert into the data structure always takes a constant time. Queuing and distribution of alerts are two independent operations. Therefore, they are performed concurrently using two different threads that shall be explained later in the implementation chapter 4. Acknowledgements are also handled in the same way as an alert. There is no difference between the queuing and distribution of alerts and acknowledgements.

Queuing:

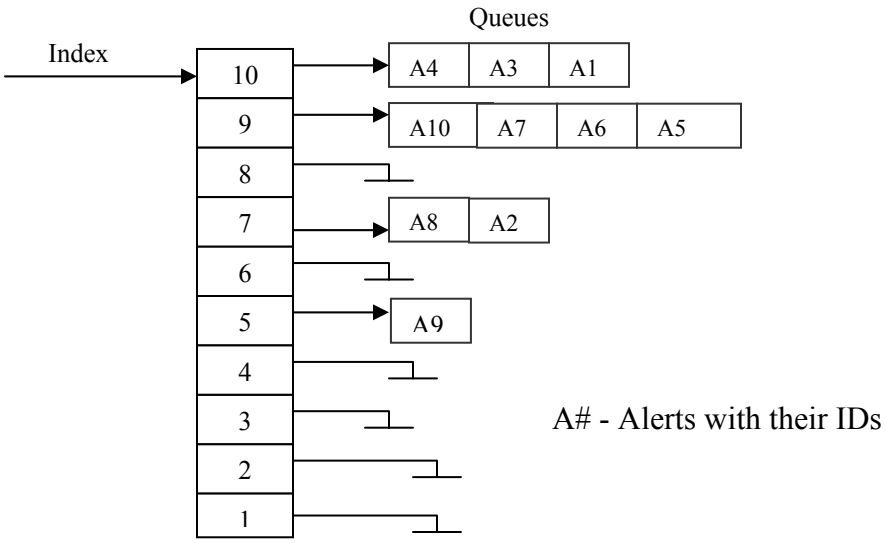


Figure 3.6. Priority Queue Data Structure.

Producers publish alerts independent of the distribution mechanism of the alerts on the Alert Server. Subscribing for an alert and publishing an alert are two independent tasks as the producers and consumers are anonymous in a publish/subscribe-messaging model. Therefore, alerts need to be queued and stored before the Alert Server can deliver them. The data structure used for this purpose is shown in figure 3.6. This data structure contains an array of queues. This has been done to help in distributing the alerts on the basis of priority as it differentiates alerts on their priority. This differentiation also helps in synchronization as different thread can access different queues simultaneously. Synchronization issues shall be discussed later in this chapter. The queuing of alerts is simple. Whenever a new alert comes in, it is indexed into the queue array using its

priority and then put at the beginning of its queue. New alerts are always added at the end of the queue in order to maintain their semantics. The storing of the alerts in priority queues array achieves the following requirements:

- Insures delivery of alerts on the basis of priority.
- Stores the alerts till a new consumer subscribes.
- Purging of alerts becomes easy, as alerts can be found easily indexing into the array.

Apart from this, distinction of the queues on priority also helps in achieving concurrency as different threads can work on different queues simultaneously. In order to achieve synchronization among different threads, each queue in the array has a mode attribute that says whether an alert is inserted or not. The alerts thus put into the queue are now available for distributing that will be described further.

Distribution of Alerts:

The distribution of alerts has to satisfy a number of requirements:

- Alerts have to be distributed using the priority associated with it.
- When new higher priority alerts arrive, they have to be distributed before lower priority alerts.
- The same alert should not be sent to the same consumer more than once.
- When new consumers subscribe to alerts, they need to receive alerts that have arrived earlier, and if they are higher priority alerts, they have to be distributed first.
- The above has to be done as efficiently as possible.

The sweep algorithm and the associated data structures described below accomplish the above. We describe the data structures, synchronization, and pre-emption methods used for implementing them. Alerts are distributed to their respective consumers by comparing their topics with the topics in the consumer table structure that is created and stored in the data structure when the consumer registers. The priority queues are

swept continuously (at the granularity of a priority level) and the alerts are distributed one at a time. It is during this sweeping of the priority queue that the expired alerts are also purged. Since the goal of distribution is the delivery of alerts on the basis of priority, the higher priority level alerts are delivered before the lower ones. Higher priority numbers indicate higher priority. Alerts of the same priority are delivered on First Come First Sent basis since the new alerts are added at the beginning of the queue and the queue is swept from beginning to the end. The priority queue data structure is swept using a sweeping algorithm.

Sweeping Algorithm

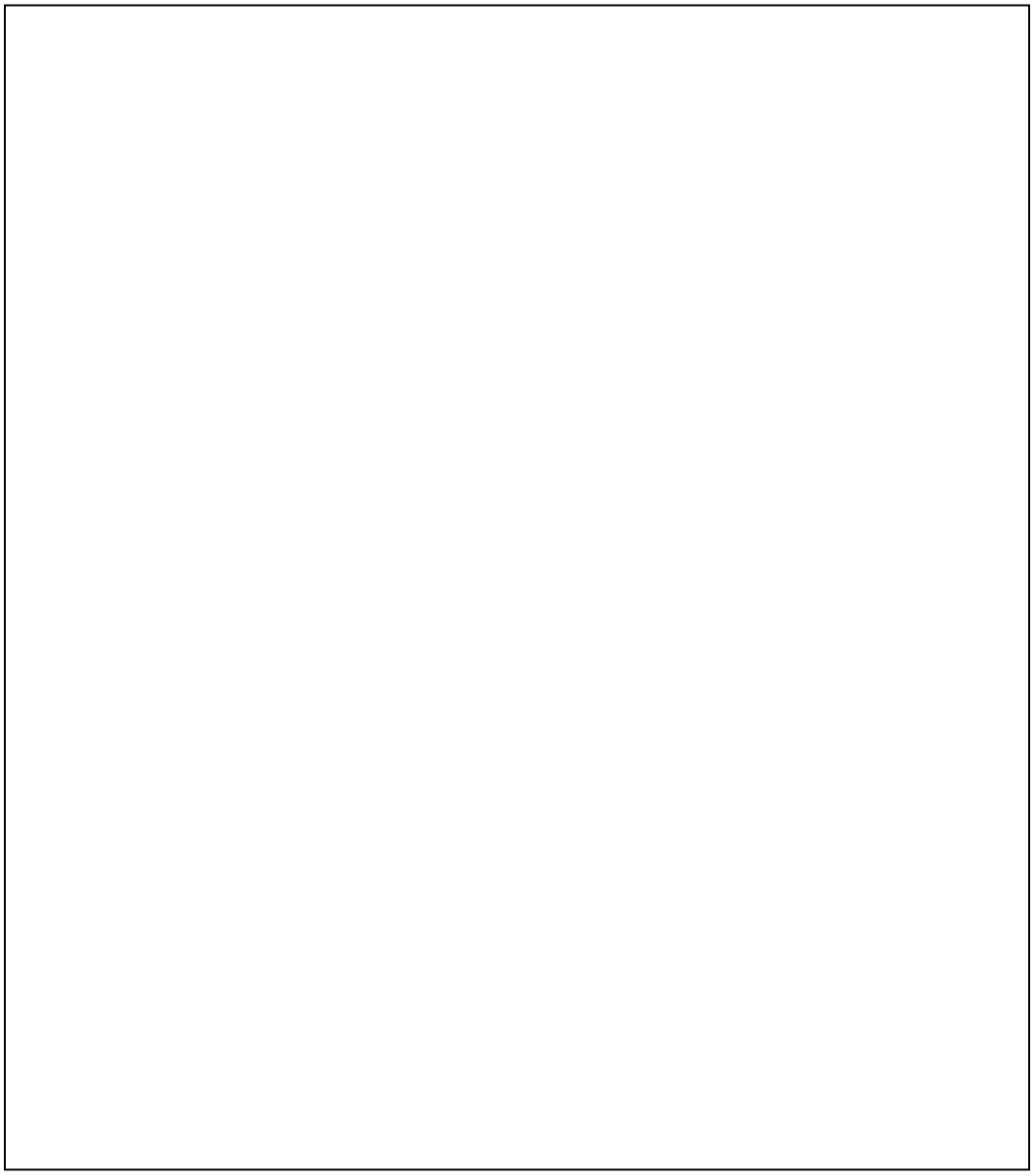
An algorithm is needed in order to sweep the priority queues. This algorithm reduces the time in sweeping the priority queue data structure and prevents the resending of the same alert to the same consumer. More over it ensures the delivery of alerts on the basis of their priority. This is maintained even in the case of addition or deletion of new consumers. The algorithm achieves this by making use of the information held in the alert hash table (i.e., alert list table attribute) and consumer lists explained in earlier sections. Alerts (i.e., Alert Topic) in the queues always map to a consumer list in the consumer table data structure in case their topic is a USER or PROFILE and to a set of lists in case the topic is a TAG. Alert list table in the alert that is created when an alert object is generated, that is accessed by the server and not by any client, holds the mapping between the consumer list and the last consumer node in the list that has received that alert. This information is necessary in trying to stop sending the message to the consumers that have already received the alert, thereby reducing the time for sweeping the consumer lists. As already explained, each consumer node in the consumer list has unique ID attribute in that list and similarly every consumer list also has a unique ID attribute in the consumer table data structure. Consumer node ID attribute helps in indicating the arrival of the consumer in that list and consumer list ID serves as an index

to the hash table in the alert message which tells that the alert has been sent till this consumer and needs to be sent to all the consumers before this consumer in the list. Since new consumers are always added to the beginning of the list, the consumers are always in decreasing order of their ID attribute as this attributes indicates the arrival number in the list. Therefore, the alert can be sent to all the consumers with the IDs greater than the consumer ID of that list in the hash table present in the alert message, thus preventing from resending the message to the same consumer more than once. Once the nodes of the lists are obtained, alerts are sent back using the information in the node. The algorithm takes the queue with the highest priority from the data structure and then traverses the queue to send each alert in it to the registered consumers. The two data structures that the algorithm sweeps may change, either due to alert addition or deletion from one of the queues or due to addition or deletion of consumers from one of the consumer lists.

```
Consumer Addition Mode
      Set all queues state to not sent.
      Get highest priority queue that has
      not been sent
      while( queue is not empty)
        runSweepingAlgorithm(queue)
        if  (Mode is same) {
            Get the next higher priority
            queue that has not been sent
        }
       else if (Mode change) {
            return to calling function;
        }
        else {
             Set mode to zero since no all
            alerts have been sent;
        }
```

Figure 3.7. Sweeping Algorithm.

Figure 3.7—*Continued.*



The algorithm executes in two phases. In the first phase the alert is checked for expiration. Expired alerts are removed from the queue. This is checked by comparing current system time with the sum of the time (arrival time) at which alert was received on the server and the time-to-live (duration) data field in the alert header. If the sum is greater than the system time then it is removed from the queue. The time received on the

server is the time when a client puts it in the request buffer on the server. But in case of persistent alerts, the alerts need to be purged from the log. This is achieved by setting the cancel bit in its log and is removed from the log when the server restarts and not immediately. Phase two consists of finding the consumer list in the consumer table. Alert destination topic is mapped to the consumer tables to find the consumer list for that alert. After finding the consumer list, then the algorithm tries sending it to all the consumers till it reaches a consumer that has last received this consumer. This works even in the case when consumers are deleted because in this case alerts are sent only to new consumers without caring about the deletion of consumers. In the absence of consumer list the sweeping algorithm continues with the next alert in the queue and applies the same two phases. After all the alerts in the queue are sent, the queue is marked as sent by setting the queue's mode attribute to DEFAULT. Once all the alerts in the queue have been sent, the algorithm starts processing the alerts in the queue with the next highest priority in case no new alerts have been added or no new consumer has subscribed to an alert of higher priority. If none of these happen, then the algorithm proceeds with the processing of the next higher priority queue repeating from phase one. But, if there is an update of one of the data structures, the algorithm should know whether a new alert is added or a new consumer is added. Therefore the algorithm needs to differentiate between alert addition and consumer addition. In alert addition case the algorithm should know whether alert has been added to the queues of higher priority and in consumer addition case, the sweeping of the queues should start afresh (with queue of priority 10), since it is not possible to know whether the newly added consumer has subscribed to high priority alert or a lower one. It should be noted that consumers only subscribe to topics and not to alerts directly. There might be a case where in a consumer might subscribe to a topic and there might be many alerts published on the same topic with different priority. Therefore in the case of consumer addition the sweeping of the data structure should start from

priority level 10. Thus, after sending all the alerts in queue the algorithm runs in alert addition mode if a new alert has been added in between and runs in consumer addition mode if a new consumer is added. It keeps running in the same mode if nothing happens till all the alerts in the queues have been sent. If during the sweeping of the queue, both addition of alert and consumer happens, then the algorithm runs in the mode that happened last. The algorithm till executes the same two phases except that in case of consumer addition mode it marks all the queues as NOT SENT by setting the mode attribute of the queue to CONSUMER ADDED (CA) before starting two phases. In alert addition mode, this need not be the case, as it can be known which queue has not been sent depending on the alert added.

Algorithm:

The Sweeping algorithm pseudo code is shown in figure 3.7. Initially queue Q with the highest priority is sent to the algorithm. Let A_0, A_1, A_2, … be the alerts in the queue. Their subscripts indicate their positions. Let $A_{current}$ be the current alert that is being distributed in the queue and A_{next} be the one after the current alert that needs to be distributed. Let PT, ST and IT be primary, secondary and internal hash tables that store the information about the consumers registered and let AHT be the table that provides the mapping between the list ID attribute of the consumer lists in the tables and the consumer node ID attribute that has last received the alert in that list. The key and the value mapping in a table is shown by HT (list[ID], consumer[ID]) where HT is a table holding the mapping. X [Y] indicates an attribute Y of an object X. For example, topic [$A_{current}$] indicates the topic field of the alert.

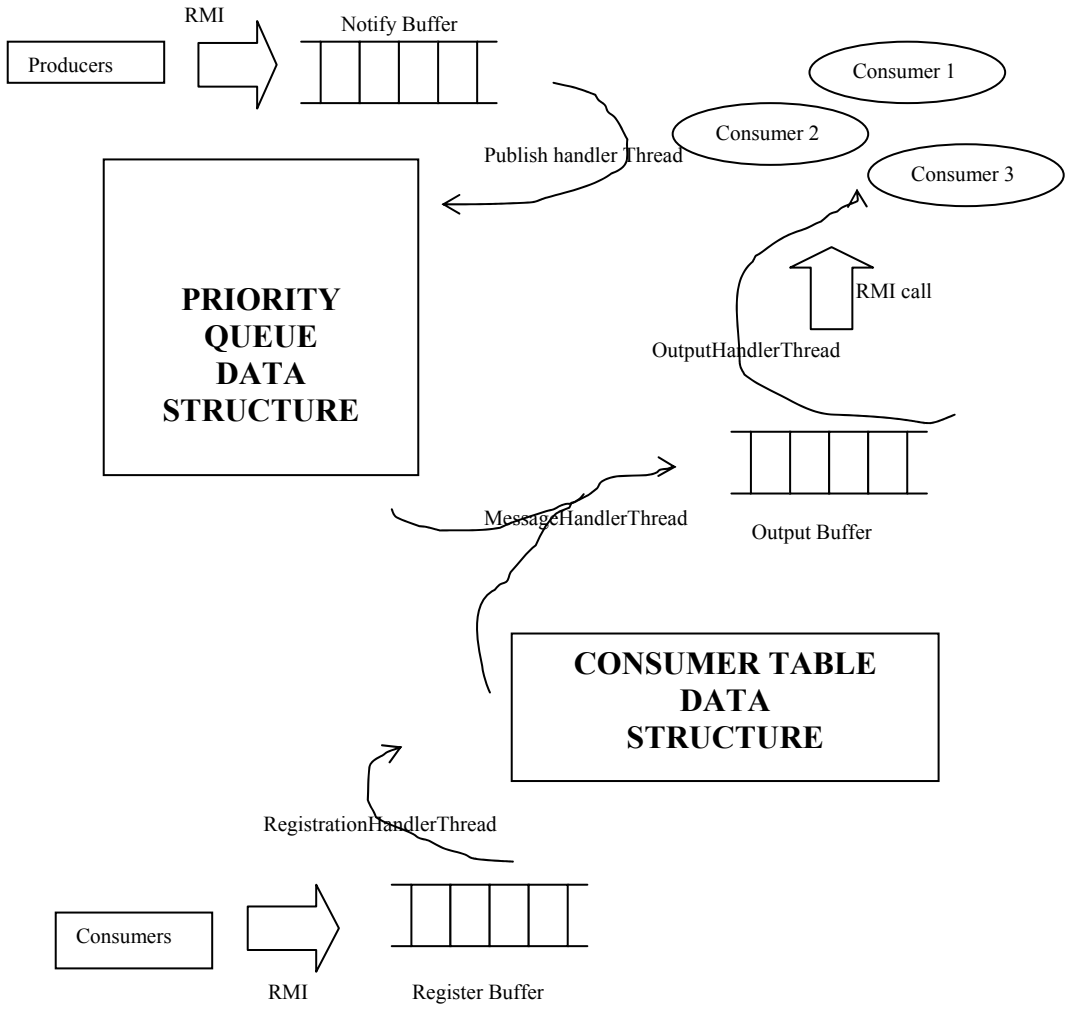


Figure 3.8. Alert Server Architecture.

3.8 Multithreading the Alert Server

The Alert Server needs to be multithreaded in order to handle clients asynchronously. The Alert Server uses Java Remote Method Invocation [RMI] in its communication interface. RMI calls are blocking therefore these calls need to be handled asynchronously. Client requests are queued. Since each request is independent and there is no guarantee that they will arrive within a certain time there is a queue for each type of

request and a different thread handles each different request. Multithreading also helps in improving the scalability of the server.

The clients put the messages in the queue and continue with their processing. Since each queue has a thread listening on it, the thread is awakened when the queue is not empty. The data structures handled by each thread are shown in figure 3.8. There are other threads for handling other requests, such as canceling an alert, unsubscribing for a topic. The threads shown in figure 3.8 are the threads that handle registration for a topic, publishing an alert and the delivery of alerts to different consumers. The publish handler listens on the notify buffer that holds the alerts that are published by different clients. It places these alerts in the priority queue data structure on the basis of their priorities. On the other hand, the registration handler handles the registration in the registration buffer independent of the publish handler. This thread constructs consumer nodes that hold consumer information used for sending the alert and puts them in the appropriate consumer lists in a consumer table data structure explained earlier. The message handler thread runs the algorithm on the priority queues and consumer tables and places the alert and its consumers in the output queue. The output handler thread picks up these alerts and delivers them to the consumers. This thread makes RMI calls to the clients to deliver the messages. The cancel and the unsubscription thread keep processing the cancel and unsubscription requests in a similarly way as publish and subscription handlers. Handling of different requests and the flow shall be explained in detail in the implementation chapter. Since this is a threaded architecture, it is essential to solve the synchronization issues. These issues are discussed next.

3.9 Synchronization Issues

The Alert Server is made up of several data structures that will be shared and hence may be concurrently accessed by threads. Following is the list of shared data structures:

1. **Consumer list:** list of register objects in the hash tables for registration. These lists are shared by registration handler thread, message handler thread and unsubscription handler thread. The registration handler thread handles the registration of new clients to the server by adding nodes to this list. The message handler thread reads the list for consumers that have registered for an alert when the thread sweeps across the priority queue. The unsubscription handler thread removes the consumers from the list when they unregister.
2. **Priority Queues:** array of ten queues that store the alerts on the basis of their priority. These queues are handled by the publish handler thread and message handler thread. The publish handler thread adds alerts to the priority queues while the message handler thread sweeps the priority queue to compare the alert topic with the consumers that have registered with the server for that topic.
3. **Output Queue:** Queue containing delivery objects that have alerts and the consumer information that is needed to send the alert to the subscribers. The message handler thread adds delivery objects to the output queue while the output handler thread delivers these objects handles this queue.
4. **Alert log file:** This file is updated by the publish handler thread and cancel handler thread. The publish handler thread writes into the file when an alert needs to be persisted. The cancel thread accesses the file when a persistent alert needs to be cancelled.
5. **Secondary Hash table:** hash table that helps to distinguish between different topics for the same alert types. These hash tables are accessed both by registration handler thread, unsubscription handler thread and message handler thread.

Race Conditions: When the result of two or more threads performing an operation depends on unpredictable timing factors, there is race condition. Example of a race condition: Unsubscription thread is in the process of deleting a consumer node at position

7 from the consumer list. Message handler thread is traversing the consumer list to get consumer node at position 13 to put it in output queue. Message Handler thread could be looking at node 7 when the list manipulation is occurring. This thread will decide that node isn't the desired node and goes to the next position in the list. However, since unsubscription thread has disconnected this node from the list the next position could be NULL. The result of what unsubscription thread reads will hence depend on the timing factor and has been compromised by the race condition. Hence the access to the consumer list and several such shared data structures must be guarded for mutual exclusion. This can be attained using synchronization mechanisms or locks. There are several types of locks and the right choice must be made.

3.10 Types of Locks

Mutex: Mutex lock is a synchronization primitive that allows multiple threads to synchronize access to shared data by providing mutual exclusion. The mutex lock has only 2 states: locked and unlocked. Once a thread has acquired the mutex lock on a data structure other threads attempting to lock the structure will be blocked until it is unlocked. Since mutex allows only one thread to access any data at a given time, it is the most restrictive type of access control. For example, when a mutex is used to synchronize access to a list, the mutex will control the entire list. While the list is being accessed by one thread it is unavailable to all other threads. If most accesses are reads and writes of the existing nodes as opposed to insertions and removes, then a more efficient approach will be to allow nodes to be individually locked.

Read-write: Read-write lock is another synchronization primitive that was designed specifically for situations where shared data is read often by multiple threads/ tasks and rarely written. A read-write lock is similar to a mutex lock except that it allows multiple threads to concurrently acquire the read lock whereas only one writer at a time may acquire a write lock. In the current scenario the *Insert* or *delete* operation on a list will

require acquiring the read-write lock in the *write_lock* mode, while the seek (search) of a node will require acquiring the lock in the *read_lock* mode. By using the read-write locks we can have search the data structure in parallel in the Alert Server. The only drawback of using read-write locks is that locking operations take more time than the locking operations on mutexes. Hence locking strategy must be chosen carefully. Read-write locks are justified for the *consumer list* and priority queue data structures in the alerts where inserting and deleting is done only once; thereafter all other operations are search operations on the list to find a particular node. *Read_lock* mode can be used to allow threads to search the list in parallel.

Semaphore: Semaphore is a synchronization primitive that has a value associated with it, which is the number of shared resources regulated by the semaphore. Whenever a thread acquires a semaphore, its count decreases by 1. Whenever a thread releases a semaphore, its count increases by 1. Any thread wanting to acquire the semaphore must wait till its count is greater than 0. Semaphores are used primarily when there is more than one shared resource that needs to be regulated.

For synchronization of data structures in the Alert Server, mutex locks or semaphores can be used when the operations involved are primarily inserts and deletes that require exclusive access. For data structures such as the consumer list, where a majority of the operations are search operations on the list and updates on individual nodes, read-write locks can be used for locking the list and semaphore or mutex locks can be used for locking individual nodes.

3.11 Other Data Structures

Apart from the consumer tables and priority queues that help in the queuing and distribution of the alerts, there are other data structures that make the server handle registration (subscription), unsubscribe, cancel and publish for an alert asynchronously. The clients make calls to the server that are blocking, hence queues are needed to handle

requests asynchronously. These data structures are basically queues in which the server stores the client requests. There is a queue for each type of interaction the clients have with the Alert Server. Each new request is added at the end of the queue. Each request is then removed from the queue and handled separately. There is a queue for each type of request from the client:

Registration Buffer

Register buffer contains requests from clients that register for an alert with the server. Buffer contains registered objects that hold information regarding the registration request. These objects are then inserted in the consumer lists of the consumer table by the consumer table data structure.

Unsubscribe Buffer

Unsubscribe buffer contains requests from clients that want to unsubscribe to an alert. This buffer contains the same register objects as mentioned above but in this case they are used to delete these objects from the consumer table data structure. Unsubscription requests have been put into a different buffer so that the server handles multiple client calls for subscription and unsubscription simultaneously.

Output Buffer

There is only one output queue on the server. So any alert to be delivered to the client is put in this queue. The sending of the alert to the client is discussed in the next section. The queue contains delivery objects that hold information required for delivery and also the alert that needs to be delivered.

Cancel Buffer

This queue stores requests from clients that want to cancel alerts. The canceling of alerts, as already explained, is decided by the cancel policy in the header of the alert. This policy decides whether only the producer or any client can cancel the alert. A separate thread called the Cancel Thread handles this queue.

Notify Buffer

This buffer stores requests from clients that publish the alerts. Alerts that are sent to the server are first put in the notify buffer and, after logging, are put into the priority queues. This buffer just contains alerts that are published. Since acknowledgements are nothing but alerts with the correlation ID set to the ID of the alert being acknowledged.

Table 3.1. Data Structures and their locks

Data Structures and Characteristics	**Locks used with Rationale**
Consumer list: list of consumer nodes that are added when a client registers and are deleted when the consumer unregisters with the Alert Server. Whenever the Alert Server has to send an alert to the clients, it scans the list.	Mutex locks are used since operations used are primarily inserts and deletes which happen when a client registers or joins. These operations need an exclusive lock mode that is provided by mutex locks**.** Using mutex locks is preferred to read-write locks; also because an operation on read-write locks have a high overhead.
Alert log file*:* file that stores the alerts that are published	Mutex locks are used here as file read and write should be mutually exclusive.
Output Queue**:** queue of alerts and consumers that are added when an alert is to be sent to its consumers.	Mutex locks are used in this case since the only two operations in this case are reading and writing. Nodes are added in the end while the deletion is done from the front.
Secondary Hash table: table of topics with their consumer lists. The table object is added only when there is a new topic.	Read-write lock for locking consumer lists. Write lock provides exclusive access while inserting or deleting nodes in the lists. When accessing list in shared (read) mode, lock hash table is used for managing access to individual lists.

Details of the workings of the threads are explained in the implementation chapter of Alert Server. The design of the Alert Server handles only JAVA clients. In order to support C and C++ clients that send alerts to the server, another server that acts as a proxy to the Alert Server needs to be implemented. The proxy needs to be designed such that it can efficiently transfer calls to the Alert Server with a minimum overhead.

CHAPTER 4

IMPLEMENTATION OF ALERT SERVER

This chapter explains the implementation details of the alert server. It first explains the communication interface of the Alert Server followed by the workings of each thread and how each thread manipulates various data structures.

4.1 Alert Server Communication Interface

The alert clients communicate with the Alert Server through the remote method calls. They can publish and cancel an alert, subscribe and unsubscribe to an alert. The next section tries to explain a little bit of Java RMI and also explains some of the reasons for choosing it over other similar technologies like CORBA and COM/DCOM [9, 10].

4.1.1 Choice of Java Remote Method Invocation

Remote method invocation allows Java developers to invoke object methods, and have them execute on remote Java Virtual Machines (JVMs). Under RMI [11], entire objects can be passed and returned as parameters, unlike many remote procedure call based mechanisms (e.g., RPC) that require parameters to be either primitive data types, or structures composed of primitive data types. Using RMI, any Java object can be passed as a parameter - even new objects whose class has never been encountered before by the remote virtual machine. This implies that new code can be sent across a network and dynamically loaded at run-time by foreign virtual machines. Java developers have a greater freedom when designing distributed systems, and the ability to send and receive new classes is an incredible advantage. More over, Remote method invocation has a lot of potential, from remote processing and load sharing of CPU's to transport mechanisms

for higher-level tasks, such as mobile agents, which execute on remote machines. Because of the flexibility of remote method invocation, it has become an important tool for Java developers when writing distributed systems. Since Alert Server has been implemented in Java, RMI has been the choice over CORBA. Moreover, CORBA [8] does not support transfer of objects or code and garbage collection. This becomes very important in this case, as the server has to handle a large amount of alerts at point in time. Apart from this, CORBA specifications are still in a state of flux. Last but not the least, not all classes of applications need real-time performance, and speed may be traded off against ease of use for pure Java systems.

4.1.2 API for communication

As already explained, there are basically six remote calls on the server that the alert clients can invoke. Clients do not make these calls directly. The clients in turn use the APIs in "MakeRMICall" class in the "alertserver.client" package. All methods in this class are static. The remote methods are wrapped by the static methods in this class, that also check for any errors in the arguments. This is necessary to catch the errors on the client side and avoid propagating them to the server.

Publish an alert:

```
public static void alertSend(Alert alertobj)
```

The alert that needs to be published is sent using the above API in the "MakeRMICall" class present in the "alertserver.client" package. This method internally calls the actual remote method on the server provided in its remote interface. This method also catches any exceptions thrown by the client. The remote method thus called puts the alert in the register buffer for processing by the server. The API explained below is remote method call in the Remote Interface provided by the Alert Server.

public void alertCancel(Alert alertobj) throws RemoteException; This API is a RMI call that passes the alert to the server. This API returns a void and takes an alert as an argument. This is also a blocking call and throws a remote exception like all other remote calls. This method call places the request in the cancel queue on the server.

Subscribe and unsubscribe to an alert:

Clients can subscribe and unsubscribe alerts by using the following methods in the "MakeRMICall" class. These methods in turn call the remote methods on the server and also catch any remote exceptions thrown by them. These methods take a registration filter, user id and alert type as its argument and return a void. The API for subscription and unsubscription is given below

public static void alertRegister (String filter, String userid, AlertType type); public static void alertUnregister (String filter, String userid, AlertType type);

The above APIs call remote methods on the server for registration. These remote methods are explained below.

public void alertRegister(String filter, String userid, AlertType type) throws RemoteException;
public void alertUnRegister(String filter, String userid, AlertType type) throws RemoteException;
These calls on the server put the register information in the register and unregister queues respectively. These requests are then handled by registration and unsubscription threads, which update the consumer data structure accordingly. These calls are blocking like all the other calls.

Cancel an alert:

public static void alertCancel(Alert alert)

Clients use the above method in "MakeRMICall" class of the client package to cancel alert. This API is used instead of calling the remote method, as it needs to check

the cancel policy of the alert. If the cancel policy is ANY then it makes the remote method call but if the cancel policy is ORIGINATOR then the sender user ID in alert is matched with the user ID of application (i.e., the client application that wants to cancel the alert). If the match returns true, then the remote method is invoked, else a message is returned indicating that the application cannot cancel the alert, as the alert was not produced by that application. This causes the computation to be done on the client side than on the server side.

public void alertCancel(Alert alertobj) throws RemoteException; This API is a RMI call that passes the alert to the server. This API returns a void and takes an alert as an argument. This is also a blocking call and throws a remote exception like all other remote calls. This method call places the request in the cancel queue on the server.

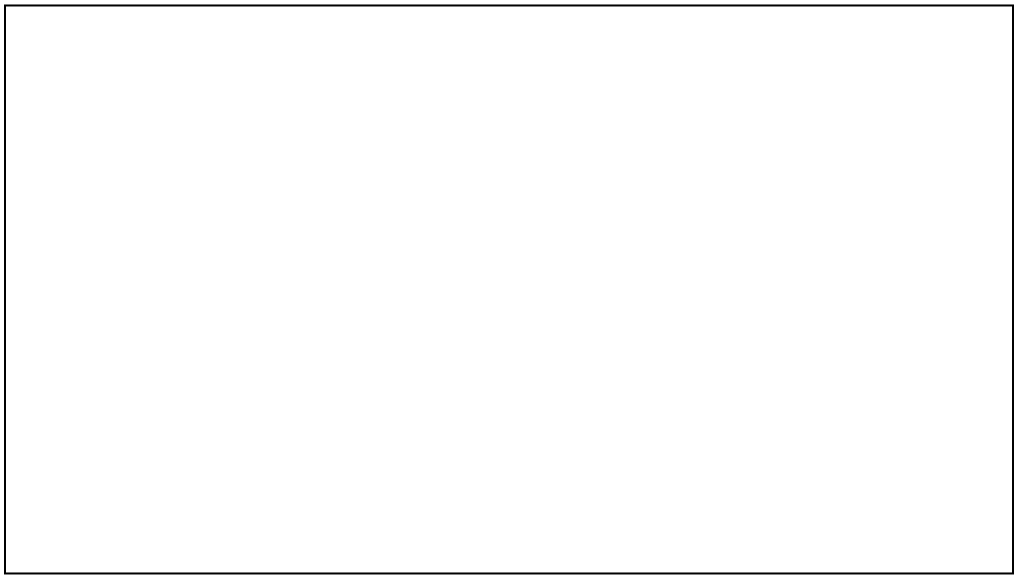
4.2 Workings of threads

Alert Server is a threaded server. As already explained, there is a thread for each request type and a thread for sweeping the data structures and finding the registered consumers for alerts. Therefore, it is essential that the threads be implemented in an efficient manner, preventing synchronization problems. The workings of the threads are explained next.

4.2.1 Publish handler thread

This thread is mainly responsible for processing the requests in the notify buffer. This buffer holds the alerts and acknowledgements that are published by the clients. This thread removes the requests from the buffer and places them in the priority queue data structure for distribution by the Message Handler Thread. After putting the alert into the priority queue, buffer log sequence number of that log (log with the same priority as that of the alert) is updated with the ID of this alert indicating this is the most recently added alert to the queue with the priority same as that of alert. But before it could wait on the

buffer for incoming requests when the thread starts initially, it reads persistent alerts from the log and puts them into the priority queue data structure. This reading is done in the beginning when the Alert Server starts. It is also during this time that the alerts that have been cancelled are purged from the log and only those that live are read and put into the priority queues. Persistent alerts are read from the logs using the index table in each log file and scanning the records of the table. Its reads the log files on the basis of their priority. The thread knows a logs priority by the name of the file. These log files are named as "persistP.dat" where P stands for the priority. The thread has to obtain the write lock of the queue before it can insert alerts into the queue and a mutex lock on the log before it could be read the log. Each insertion of an alert results in this thread notifying the message handler thread to run in ALERT ADDITION MODE (AA). It indicates this by setting a common object between the message handler, registration handler and itself to AA and setting the mode on the queue it is inserting the alert into also to AA. This is necessary, as the message handler should know that an alert has been inserted into the queue. This is done by obtaining the locks on the both the common object and the queue. The logic used by the thread is shown



This chapter explains the implementation details of the alert server. It first explains the communication interface of the Alert Server followed by the workings of each thread and how each thread manipulates various data structures.

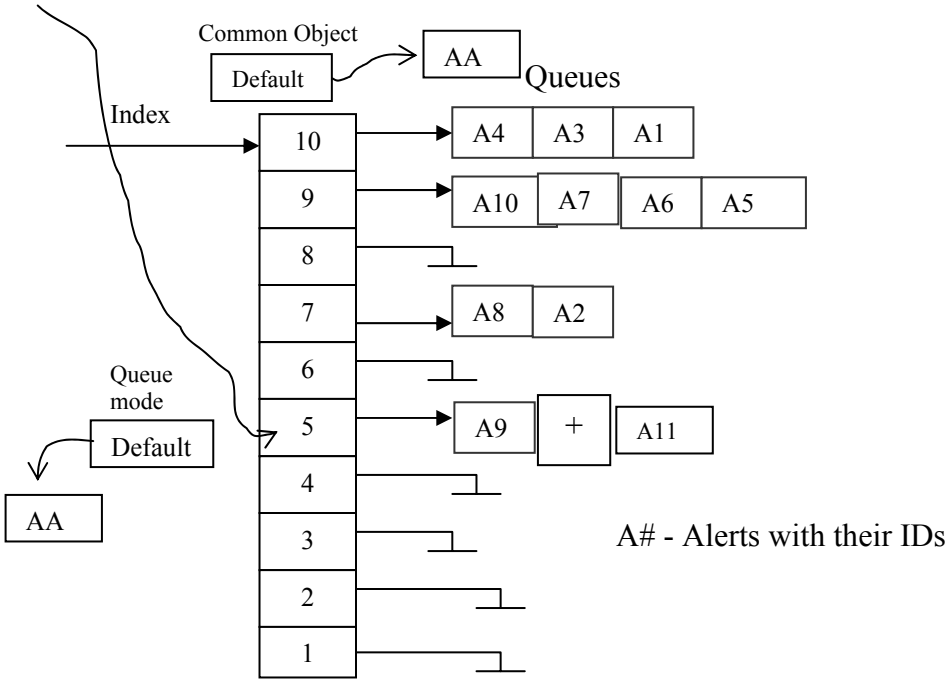


Figure 4.1. Priority Queue data structure in Alert Inserted Mode.

Each insertion of an alert results in this thread notifying the message handler thread to run in ALERT ADDITION MODE (AA). It indicates this by setting a common object between the message handler, registration handler and itself to AA and setting the mode on the queue it is inserting the alert into also to AA. This is necessary, as the message handler should know that an alert has been inserted into the queue. This is done by obtaining the locks on the both the common object and the queue.

The working of the thread can be better understood with an example. Assume that the state of the priority queue array is as shown in figure 4.1 with the common mode and the queue mode as default. This figure shows the addition of alert A11 to the queue with priority of 5. Initially when there are no new alerts and no new consumers, the message

handler is waiting for some new alerts. When a new alert is added to the queue, the message handler is notified by setting the common object mode to AA. The publish handler also sets the mode of the queue in which it inserts the alerts to AA so that the message handler knows that the alert has been added to this queue and it has to process the queue when it sweeps the queues from their highest priority.

4.2.2 Registration handler thread

```
public void run( )

 while (!interrupted) {
   if (registration queue is empty)
          wait
   Remove request from queue.

   Update the consumer table data structure by adding the consumer info.

    Notify the message handler thread to run in CONSUMER ADDED MODE
}
```

This thread handles the requests for registration from clients. These requests are queued in the register buffer. It removes the requests from this buffer and places them in the consumer table data structure in a consumer list. The execution of this thread is shown above. It updates the consumer table upon receiving a new consumer subscription and notifies the message handler thread to run in CONSUMER ADDED (CA) mode. This handler after adding the consumer to the consumer list as shown in figure 4.2, updates the added attribute of that list by one and sets the consumer ID attribute of the consumer newly inserted to the added attribute that has recently been updated. It then notifies the message handler to run in the CA mode by setting the common object to CA. It cannot set the queue mode to CA as it cannot know to what alert the client has subscribed. It should be noted that clients could subscribe only to topic and not to alerts directly. It this reason that causes the message handler to run in two modes.

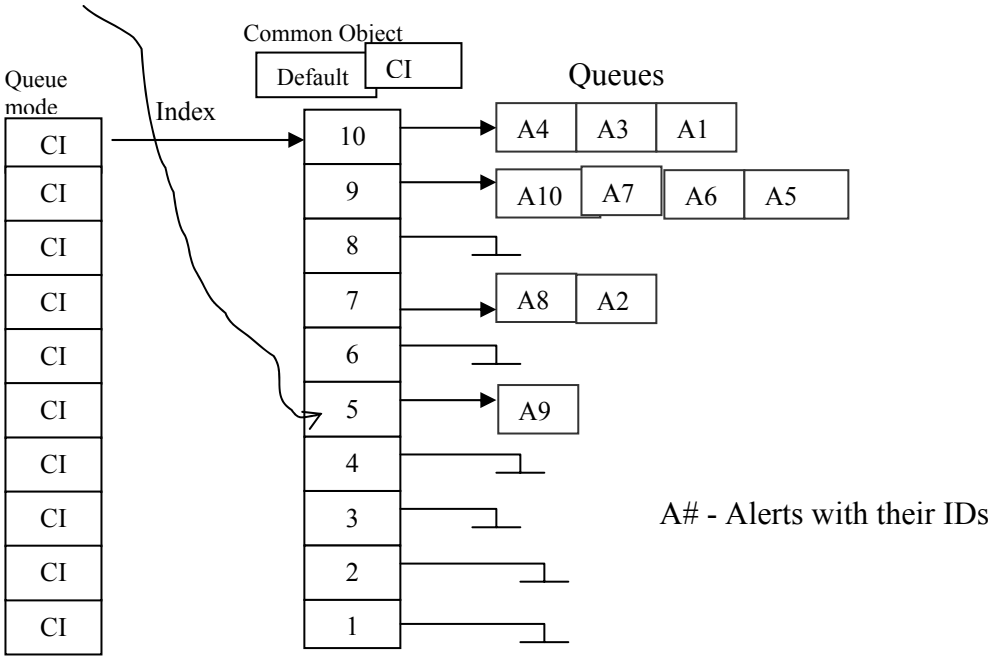


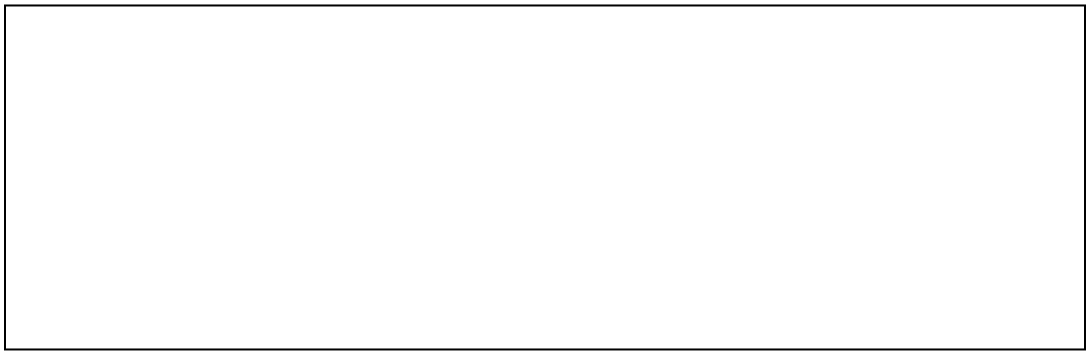
Figure 4.2. Priority Queue data structure in Consumer Added Mode.

4.2.3 Message Handler Thread

This thread is the main thread that helps in distributing the alerts. It continuously sweeps the priority queue and consumer table data structures for distribution of alerts. The details of the priority queues and consumer table data structures have been explained in chapter 3. This thread therefore has to access two data structures, priority queues and consumer tables that are being updated independently by two other threads. It needs to run whenever there is addition of alerts to the queues or addition of consumers in the consumer lists to the priority queue array and the consumer table data structure. In order to indicate the change in these data structures to this thread and synchronize the access of these data structures with the other threads it is essential for a different synchronization mechanism. Other threads (registrations and publish handler) communicate with this thread using a common indicator object, informing the thread that an alert has been added

or a consumer has been added. This indicator object will help the thread decide whether the thread has to run in ALERT ADDED MODE or CONSUMER ADDED MODE. This is essential in order to ensure the delivery of the alerts on the basis of priority as a higher priority alert might have arrived or a new consumer might have been added when this thread is processing a lower priority alert. This differentiation is needed, as the thread has to follow different logic in two cases. It has to wait when there are no more alerts to send and no new consumers added. It runs in ALERT ADDED MODE, whenever an alert/acknowledgement is published and the priority queue array structure is updated by the publish handler thread. It is asked to run in this mode by publish handler thread by setting the mode of common indicator and mode of queue in which the alert is being inserted to ALERT ADDED (AA). It keeps on running in this mode till all the alerts in that particular queue, not the entire priority data structure, has been sent or a new consumer is added. If no consumers are added meanwhile, it continues to run in this mode else it starts running in CONSUMER ADDED MODE (CA) as the new consumer might have subscribed to an alert of higher priority. The mode is changed only when there is a new alert added or a new consumer added. If both these cases happen simultaneously, then the thread runs in a MODE that was last set. It marks the queue as "DEFAULT" when all the alerts in that queue are sent. This indicates the thread that the queue has been processed when it tries to sweep from next time. The thread keeps running with the next highest priority queue whose mode is not marked default when the mode is not changed. In case of consumer insertion, the register handler notifies the message handler to run in CA mode by setting the common object to CA. The message handler then locks the entire data structure and sets the mode of all the queues to CA. This is essential since the threads do not know the alerts the new consumers have subscribed for. Therefore message handler has to sweep the queue from the highest priority. After doing this, a new alert might have been added to a queue changing its

mode to AA. Message handler sweeps this when its turn comes since the mode is not DEFAULT. It should be noted that the sweeping is always done from the highest priority and these modes of the queue only tell the thread that it needs to send the alerts in the queue. A mode of default for a queue indicates that all alerts have been sent or there are no alerts in the queue. The running of Message Handler thread in the ALERT ADDED MODE is shown below



In this mode the thread just starts sweeping the entire priority queue data structure picking one queue (i.e., one priority level) at a time on the basis of their priority and running the sweeping algorithm. The thread keeps on running in this mode until a new consumer has been added. The granularity of running the sweeping algorithm has been fixed to a queue instead of an alert to reduce the time needed to send the alerts as getting a lock of entire queue is faster than waiting for lock of each alert. So, whenever a new consumer subscribes the consumer table data structure is updated by the registration handler thread that notifies the Message Handler to run the algorithm in CONSUMER ADDED MODE. Also, overhead associated with checking for new alerts and consumers is likely to be high. The running of Message Handler Thread in CONSUMER ADDED MODE is shown below:

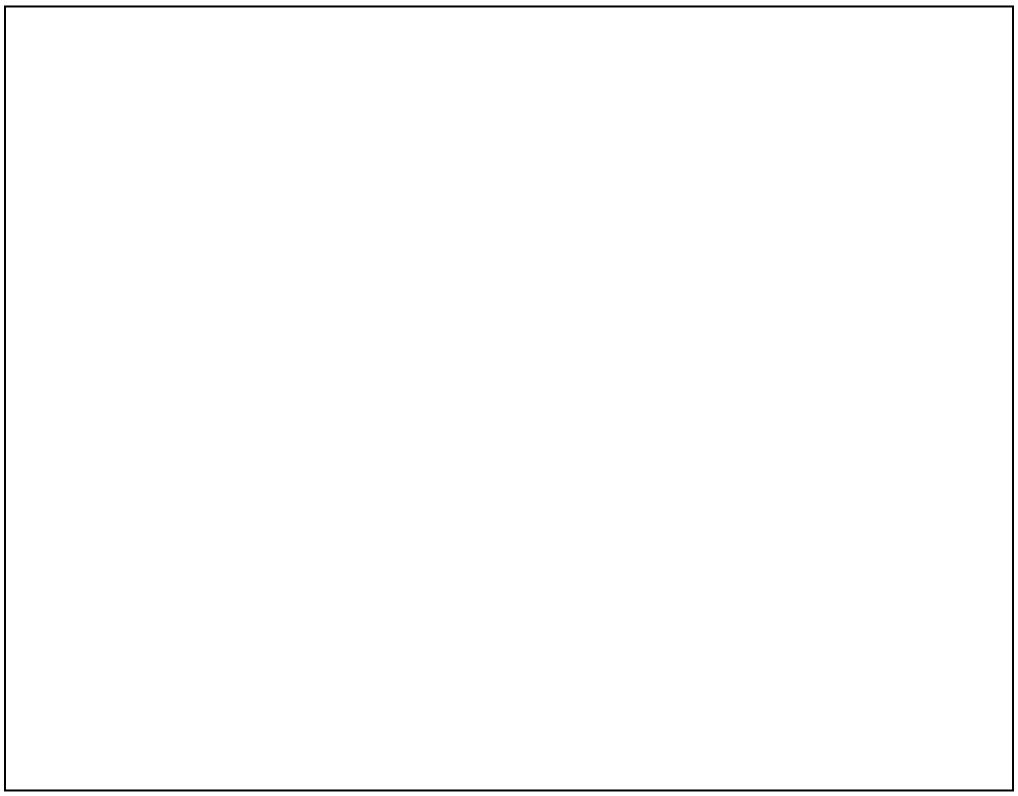
```
private void sendtoOQbyMode2(Indicator indicator) throws InterruptedException

    if(priority queue data structure is empty)
            return to calling function and wait

    Set all queues state to not sent.

    Get highest priority queue that has not been sent

    while ( queue != null) {
      runSweepingAlgorithm(queue)
      if  (MODE IS STILL CONSUMER ADDED)
         Get the next higher priority queue that has not been sent
      else if ( ALERT MODE ADDED)
         return to calling function;
      else
         Set mode to zero since no all alerts have been sent;
```

Delivery of alerts to consumers by Message Handler Thread:

This section explains the delivery mechanism used by the thread in order to find consumers in the list. The working of the thread is shown above and the mechanism is explained further.

Alert Server needs to send the alerts to their consumers in an efficient manner. A normal sweeping of the consumer tables where the consumer information is stored, is not sufficient since the consumers can register and unsubscribe to an alert dynamically. A sweeping algorithm is used for this purpose. The details of the sweeping algorithm can be found in chapter 3. Each list in the consumer table data structure has a unique ID. Each list contains consumer nodes that have unique attribute that indicate their arrival in the list. Apart from this, the list has two other attributes, added and deleted, to keep track of consumers that are added and deleted dynamically. The added attribute increases by one when a new consumer is added due to registration and the deleted attribute increases by one when there is a deletion of a consumer due to unsubscription. But these attributes of the list alone are not sufficient; alerts also need to keep track of the last consumer that has

received it to reduce traversal of the entire consumer list. For this purpose, there is a hash table called alert list table in every alert, which stores the list ID and the consumer arrival attribute of the consumer node. This tells the alert about the consumer that recently received it. This table always has a single entry in case of alerts that have USER and PROFILE topics since topics always map to a single consumer list in the consumer table data structure while the alert list table may have multiple entries in case of TAG since the subscription is based on regular expressions.



This thread after finding the consumer list for an alert first checks whether this alert needs to be sent for the first time by checking the number of consumers that received it by using the entry of this list in the alerts list table. If the alert needs to be sent for the first time, then it needs to be sent to all the consumers in the list irrespective of how many consumers have been added or deleted till that point. But if the alert has

already been sent and needs to be sent next time during sweeping, then there are five possible scenarios to be considered. In order to stop resending of alerts, it uses the other list attribute, added and the list table in the alert. In order to understand these scenarios, consider that the alert has already been sent to consumers C1 and C2. Therefore added (A) and deleted attributes (D) shall be 2 and 0 respectively and the entry in the alert table (AL) for this list will be 2 since C2 is the consumer that has last received this alert. It should be noted here that consumers are always added to the front of a list. These scenarios have been explained below with the help of the table 4.1.

Scenario 1: Addition of new consumers

This is the simplest case where only new consumers have been added. The thread has to send to all the consumers from the beginning in the list till it reaches the ID of the consumer that last received the list. This is stored in the alert when it was swept the last time. For the initial scenario above, let us assume that consumer C3 has been added. The list attribute A changes to 3 and attribute D is unchanged at 0. The thread sends alert to all the consumers whose IDs fall between A and AL (2) thus avoiding resending and update AL to 3 since C3 is the last consumer that received the alert.

Scenario 2:Deletion of consumers with no additions

In this case, the thread should not send this alert to any consumer. The thread can know this by checking the list entry in the alert table with the list added attribute. For initial scenario given above, lets say that consumer C1 has been deleted. At this point, A is unchanged at 2 and D changes to 1. Since A is not greater than AL (2) alert is not sent to any consumer. AL is still 2 in this case as the last consumer receiving the alert is C2.

Scenario 3:Addition and deletion of new consumers

For the initial scenario, consider the addition of consumers C3, C4 and later followed by the deletion of consumer C4. Now for this case the attributes A will be 4 and D will be 1 while AL is still 2. The thread now checks AL with A and since A is greater than AL, the

alert is sent to all consumers between A and AL. This works fine for this example. But what happens when all the consumers added is deleted. In this case, A will be 4 and D will be 2. Now since A is greater than AL, it tries to send all consumers between A and AL like before, but does not find any consumers as they have been deleted. Therefore, an additional check comparing the first consumer ID in the list with the AL is necessary. If ID is less than or equal to AL, then list is not scanned since all the alerts have been sent to the consumers in it.

Scenario 4:Addition of consumers and deletion of old consumers

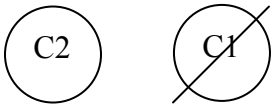
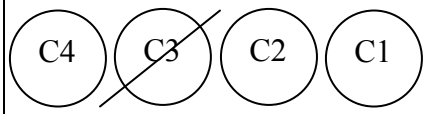
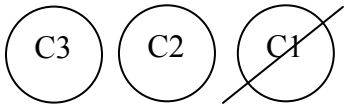
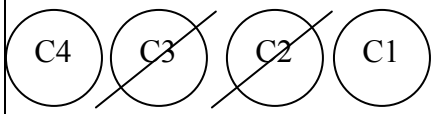
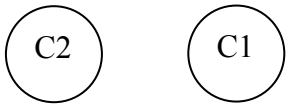
In this case consider the addition of new consumer C3 and deletion of consumer C2. Now, A is 3 and deletion is 1 while AL is 2. Here again, since A is greater than AL, thread sends alert to all consumers between A and AL.

Scenario 5: Addition of consumers, deletion of old and new consumers

Here, say there is addition of 2 consumers C3, C4 and deletion of C4 and C2. The attributes at this state shall be A (4), D (1), AL (2). Since A is greater than AL, the thread sends alert to all alerts between A and AL. While sending to consumers it also checks for the consumer ID in the list with list entry in the table. If this ID is less than list entry, then it has sent to all consumers in the list and stops any further sending.

Table 4.1. Scenarios in sweeping consumer lists

Scenarios	Consumers	AL	A	D
Initial	C2 C1	2	2	0
Addition of new consumers	C3 C2 C1	2	3	0

Deletion of consumers with no additions		2	2	1
Addition and deletion of new consumers		2	4	1
Addition of consumers and deletion of old consumers		2	4	1
Addition of consumers and deletion of old and new consumers		2	4	2
New Alert		0	2	0

4.2.4 Unsubscription Handler Thread

This thread runs in the same way as the register handler thread except that this thread handles requests for unsubscription to an alert and an acknowledgement. This thread removes unsubscription requests queued in the unsubscribe buffer and removes the consumers from the consumer table. The thread needs to obtain a write lock from the list from which the consumer needs to be deleted. This thread executes the following pseudo code:

```
public void run()
   while(!interrupted())
      Remove request from the queue

      Delete the consumer node from the registration table

      Update the deleted attribute of the list accordingly.
```

4.2.5 Cancel Thread

```
public void run( )
  while(!interrupted( ))
     Remove alert from the cancel queue
     if(alert is persistent)
        Update the table record in the log
```

Cancel thread is responsible for removing the alerts that are cancelled from the priority queue. This thread removes alerts from the priority queue and also sets the cancel bit in the log so that the alert be purged from the log when the server restarts.

4.2.6 Output Handler Thread

```
public void run( )
   while(!interrupted( )) {
      Get the delivery object from output queue
      Get alert from the delivery object
      Get the consumer from the delivery object
       if(JavaClient( ))
         SendtoJavaClients(alert, consumer)
       else
         SendtoProxyClients(alert, consumer)
       Update DLSN attribute in the log
    }
```

The working of this thread is shown below. This thread is responsible for transferring the alerts to the registered from consumers. The Message handler thread that finds the consumers, places the alerts along with its consumers in the output queue. This thread takes delivery object i.e., alerts and its consumers from the queue and sends them back to the client by making RMI calls to the client. But this thread has an additional responsibility. It has to distinguish between proxy clients and alert server clients i.e., C/C++ clients and Java Clients. In the case of Java clients, the user ID of the consumer that is present in the consumer object is a single string and in the case of proxy clients the user ID is a concatenation of the user ID of the client and the user ID of the proxy server

with a "/" in between. This thread also sends receipt to the producer of the alert once an alert has been delivered to the consumer and updates the DLSN of the log file. The receipt is sent to the producer in the same remote call that transfers the alert to the client. The user ID and Sender's IP attribute in the alert identify the produce.

4.3 Implementation of Locks

The locks used in accessing different data structures have been explained in chapter 3. This section explains the implementation of these locks. All locks belong to the *alertserver.locks* package. The locks in this package provide three different types of synchronization protocols. They are:

1. Sync: acquire/release protocols
2. Channel: put/take protocols
3. Executor: executing Runnable tasks

Alert Server uses only one protocol, the sync protocol. All the locks, Mutex, ReadWrite and Semaphore locks in this protocol implement Sync interface. This interface provides three methods, *acquire()*, *release()* and *attempt()*, which the locks override. The first method *acquire()* is used when a lock is needed to be acquired. It is essential when a thread needs to enter a synchronized block. The thread that enters the critical section or synchronized block (in JAVA jargon) needs to release the lock to let other threads waiting to enter the critical section. The *release()* method is used for this purpose. The other method *attempt()* is used to acquire a lock within a specified amount of time. Read/Write locks come with the facility to control the number of readers and writers. It also provides mechanisms to assign priorities to readers and writers. Alert Server does not need locks with such priority. Read Write locks in Alert Server are used only for issuing read locks and write locks. It should be noted that the locks in this package are non-reentrant meaning the thread that owns a lock has to wait for that lock till it releases. The relationship between different locks in this package is shown in figure 4.3.

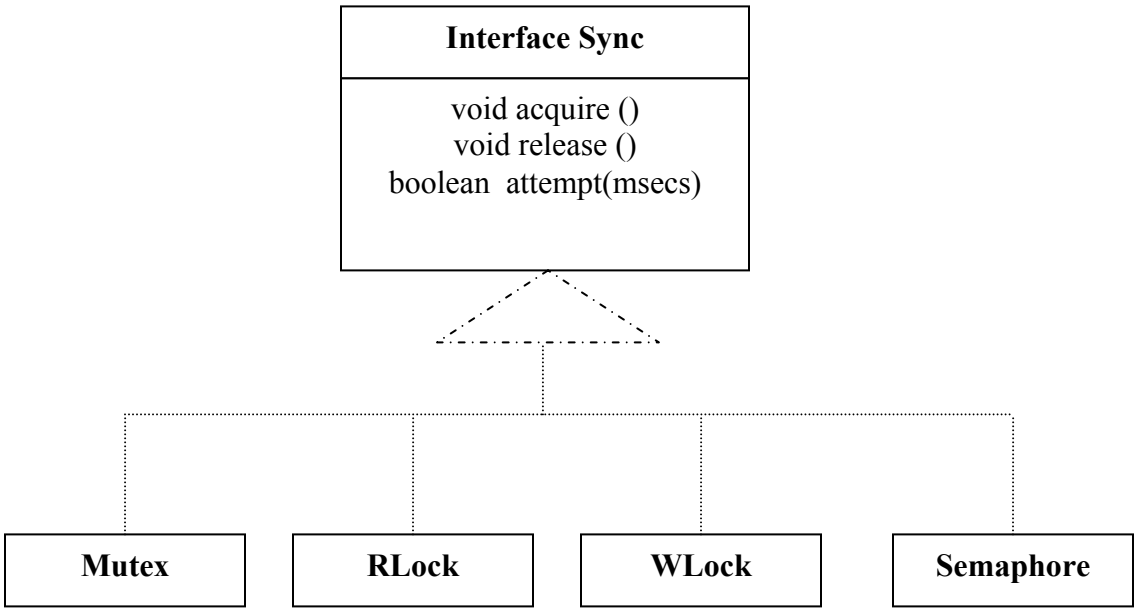


Figure 4.3. Class diagram of lock package in Alert Server.

CHAPTER 5

DESIGN AND IMPLEMENTATION OF PROXY SERVER

This chapter discusses our design of a "PROXY SERVER". The need for proxy server arises for handling C and C++ clients. In addition to multithreading, it is also necessary that proxy server handle the communication in a simple and efficient manner providing the bare minimum needed.

5.1 Design Issues

The proxy server was designed to monitor events in a distributed application environment where the client applications are producers and/or consumers of alerts. This was mainly designed so that C and C++ clients can talk to the Alert Server written in Java. Therefore clients in this chapter correspond to C and C++ clients and server refers to the proxy server unless specified otherwise. Consumer clients make RPC calls to register and unregister for alerts with the server. Similarly producer clients make remote calls to proxy server to send alerts to the Alert Server. Consumer clients also make similar calls to server to receive alerts when the Alert Server sends them. Remote Procedure Calls have been used in the communication between the client and the server. An RPC call is synchronous and blocking. The mechanism of synchronous RPC [4] is shown in figure 5.3. This means that a client making an RPC call to the proxy server is blocked till the call returns. The RPC request service procedure is in the main thread of the server process. Therefore, even if two or more clients are making procedure calls at the same time they will be handled serially by the server, and a client may have to wait for service till the server finishes servicing the previous client request. This wait will be

significant when the server is handling several clients and the events being delivered are large. In order to make the proxy server scalable it should be able to handle multiple client requests concurrently. Hence the first design goal is to have a multitasking server, as shown in. Another design goal of proxy server is that it should just provide a store and forward mechanism. It should store the requests from clients and forward them to the Alert Server. It should act as a mediator between the Alert Server and its clients. There should not be any distribution logic as Alert Server is already doing it. Clients should be handled transparently i.e., the client should assume that it is talking to the Alert Server. Proxy Server should just serve as place where in the requests and responses from the clients and Alert Server are converted into a language the other understands.

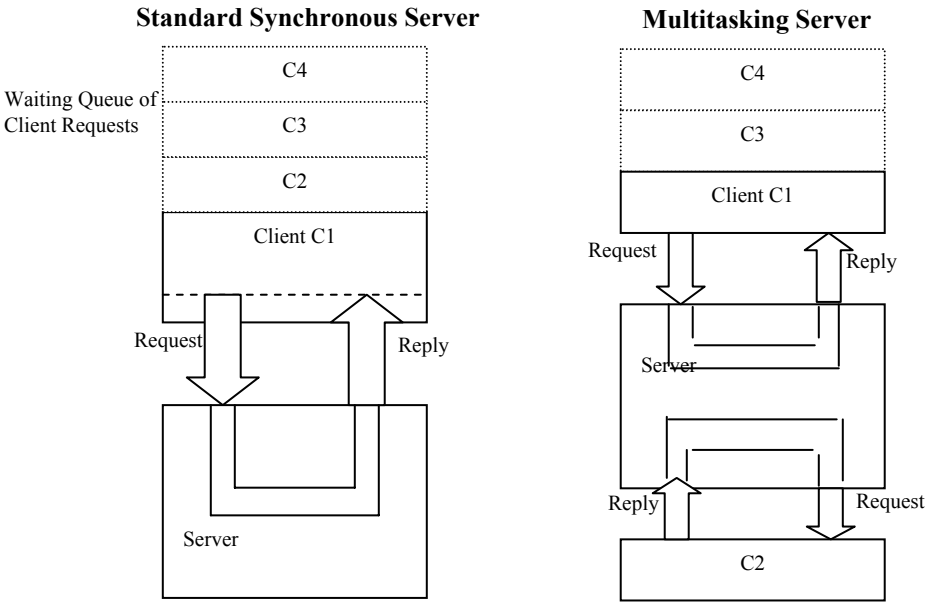


Figure 5.1. Multithreading vs Multitasking.

5.2 Architecture of Proxy Server

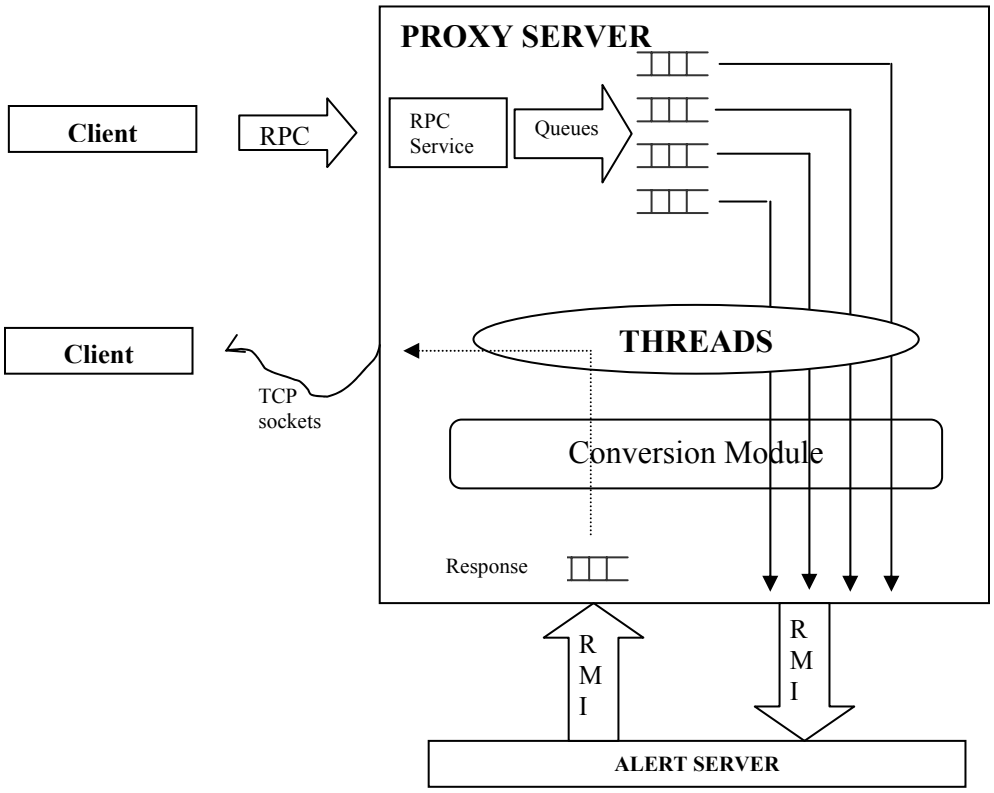


Figure 5.2. Architecture of PROXYSERVER.

C and C++ clients send their requests to this server, which in turn, forwards it to the Alert Server. Thus the architecture of the proxy server should support multiple clients asynchronously since the rate at which the request comes is not known. This is achieved by multithreading the server. It need not involve any distribution mechanism as the Alert Server already provides it. It only needs a store and forward mechanism to move alerts in either direction between the Alert Server and the clients. Therefore the data structures used for this purpose need not be complex. Simple queues solve this purpose. It serves as a place where the needed conversion from Java to C/ C++ objects and vice versa takes

place. Java Native Interface [17] has been used for this conversion. The details of the implementation and conversion are explained in the implementation section.

Before we elaborate on the implementation, it is essential to understand the architecture of the proxy server to how this has been implemented. This is shown in figure 5.2. The clients send their requests through RPC calls. There is a service procedure for each type of request. These RPC calls are blocking [4]. Since the server just stores and forward messages, queues have been used as the data structures to hold them. This also helps in allowing the client to make asynchronous calls. Each request type has its own queue, as the remote call just needs to put them in their respective queues and return. There is a queue for registration, unsubscription, cancellation and so on. Individual threads that help in propagating the requests to the Alert Server handle each queue. Since different threads share the queues it is necessary that the race conditions be handled properly. The threads take out each request from the queue and convert them into Java Objects in the conversion module and pass it to the Alert Server by making Remote Method Invocation [11] calls. The conversion module is explained in detail in the implementation chapter. The queues on the proxy that help in handling the requests and the locks used for handling race conditions and other synchronization issues are discussed below. This is can be better understood by considering different requests one at a time.

5.2.1 Publish an alert

The registration thread handles this request. New requests are always added at the end of the queues and the requests are always handled as they come. Since a node is being removed by one thread and added by another thread and there is no reading of the queue at any point of time, the lock used for synchronization in this case is a mutex lock.

5.2.2 Canceling an alert

This request is handled by cancel thread. In this queue too, new requests are added at the end of the queue and requests are handled as they come on a first come first server basis. Since only reading and writing is involved the lock used is a mutex. Since for canceling an alert, it needs to be passed to the Alert Server, this is passed in the same way an alert is passed. But just doing this, is not sufficient. There may be cases where in the cancel request is on the proxy server and the cancelled alert is in the response queue of proxy server. These responses should not be sent to the clients as they are already cancelled. This is achieved by checking the presence of the object in the cancel queue before dispatching the response to the client.

5.2.3 Subscribe and Unsubscribe to an alert

These requests are handled by subscription and unsubscription thread respectively. Like the previous queues, synchronization is handled by mutex as only writing into the thread is involved. The request pending problem happens in case of unsubscription. This is taken care in the same way as canceling an alert.

5.2.4 Acknowledge an alert

Acknowledging an alert just like sending an alert with the correlation ID of the acknowledgement alert set to the ID of the alert to which it is an acknowledgement. Alerts are handled in the same way as an alert. Therefore, they are stored in the same subscribe buffer.

5.3 Implementation Details

This section discusses the implementation details of the proxy server. First, the alternatives available to achieve the scalability and other design issues discussed above are dealt and then the APIs needed by the clients to communicate with the server are discussed. The execution of the threads in the server is discussed next. Finally, the

handling of the communication between the proxy and JAVA alert server using RMI and JNI is explained.

5.3.1 Multithreading the server

In order to achieve scalability and handle clients asynchronously, it is essential that the server support multitasking. This is achieved by multithreading or forking a child process. The proxy server uses RPC and socket protocols in its communication interface. In order to listen for client requests when the server starts running, the server process first registers the program, procedures and version numbers with *registerrpc* command. The port mapper then advertises the availability of the RPC address so that interested clients can open a channel with the server. The server then goes to sleep while the *svc_run* call listens to the other end of a socket for a client request to come along.

Based on the client request (argument), it executes one of the procedures mentioned in the *registerrpc* call and returns the reply (result) to the client. The *svc_run* routine is the heart of the server. Typically, it loops indefinitely, checking a set of socket descriptors. When it gets a service request, it switches to the associated procedure on examining the type of request. Once the procedure is executed, it loops back and waits for additional requests. Details of *svc_run* are in [4]. In order to make the server **scalable** it must execute each procedure in a thread, while the main thread (or process) listens for additional requests. Multi-tasking the server is useful mainly when there are multiple client requests that take widely different times to process. Even when the server is stuck in the middle of a request that is a long processing task, the server should be pre-empted by other brief or higher priority requests.

Either forking a process or allocating a thread to handle each request can achieve Multitasking. In order to handle the request in a separate process, a child process can be forked during dispatching of the request. Another alternative would be to start a child process for each service procedure when the server is first started. In order to use

multithreading instead of child processes, the *svc_run* routine must create a pool of threads where an idle thread will be allocated to handle a particular request. The *rpc_control* utility function provided by the RPC library provides an option of starting the server in the AUTO_MT mode wherein the procedure dispatch routine in *svc_run* allocates a thread to handle each service request.

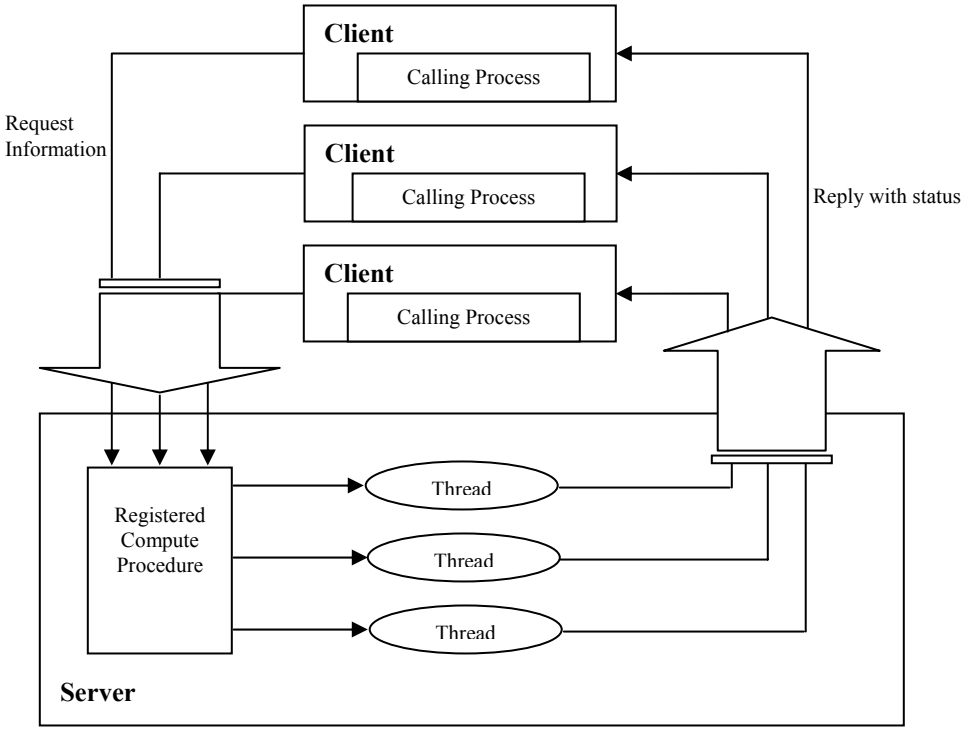


Figure 5.3. Details of thread handling.

A thread is a single flow of control within a process. Threads share a single address space. Each thread shares the resources of the parent process. Although multitasking can also be achieved by creation of child processes, multiple threads of execution provide much higher performance as compared to full-blown forking. First, since threads share global variables, memory sharing is not an issue with threads. Second, while context switching among threads, only pointer to the thread's stack and registers

needs to be saved. For process context switches, all registers, stack, data, program counter as well as several runtime state parameters of process need to be saved. Hence context switching for threads is a lot cheaper than for processes. Third, when processes synchronize, they usually have to issue a system call, a relatively expensive operation that involves trapping into the kernel. But thread synchronization is usually handled by the runtime thread library, and is less expensive as it does not require a trap to the kernel. For the above reasons multithreading was seen to be the better of the two alternatives to achieve multitasking. The threads created by the RPC service place the client requests in separate queues depending on their request types. Different threads that convert them into JAVA objects and pass them on to Alert Server by making remote calls, then process these requests. Similarly, the response from Alert Server is also queued. This response is processed by a separate thread by converting it into C structures in the conversion module and then sent to the clients. The threads have been implemented using POSIX [18] standards so that are portable.

5.3.2 Communication Interface

C/C++ clients send different requests by making RPC calls to the proxy server. As already explained, the clients may send an alert, cancel an alert and also subscribe or unsubscribe to a topic. The alert in case of C and C++ clients is a structure containing all the fields that have been explained previously. For any sort of communication with the proxy server, clients need to obtain a CLIENT handle [4] that should work with TCP since the connection that the proxy server can accept is a TCP connection. A TCP transport connection has been used instead of a UDP, as UDP RPC messages are at most 8K bytes, so procedures requiring or returning more than 8K bytes of encoded data should use TCP. The following API obtains the client handle.

cl_handle = clnt_create("proxyserver", PROXYSERVER, VERS, "tcp");

clnt_create( ) creates the CLIENT structure for the specified server host, program, and version numbers. The first argument is the name of the host; the second and third arguments are program number and version numbers respectively. The last argument specifies a connection. This should be set to TCP.

The APIs used for communication with the server are explained below. All the APIs below need a CLIENT handle.

Send an alert:

Alerts are sent by passing the alert structure along with the client handle in the remote procedure call. The API used for sending alert is

alt_send_1(alert, cl_handle);

alt_send( ) makes RPC call to the proxyserver and places the alert in the publish queue. Queue details have been explained in the previous chapter.

Alert that is sent is placed in the publish queue. A thread listening on the queue picks up the alert and forwards it to the alert server for distribution.

Cancel an alert:

alt_cancel_1(alert, cl_handle);

alt_cancel_1( ) passes the alert that has to be cancelled from the client to the proxyserver, which places it in the request queue. The first argument is an alert structure.

An alert is cancelled by passing the alert structure to the proxy server. This API is similar to alt_send_1() except that this API places the alert in a cancel queue. The API used for canceling an alert is shown above

Acknowledging an alert:

Acknowledging an alert is basically sending an alert whose correlation ID is set to the ID of an alert. Acknowledgements are also queued in the publish queue and handled like alerts.

alt_acknowledge_1(alert, cl_handle);

alt_acknowledge_1( ) passes the acknowledgement to the proxy server and places it in the publish queue for further distribution.

Subscribe and Unsubscribe to an alert:

Clients subscribe and unsubscribe for alerts by specifying a topic. As explained earlier, clients can subscribe and unsubscribe to an alert or acknowledgement. Therefore, it is essential to pass alert type, the topic and the clients name for requests of this type. This information is a REGINFO structure. The API used to subscribe and unsubscribe is shown above.

alt_register_1(reginfo, cl_handle);

alt_register_1( ) puts this structure in subscribe queue. Subscribe thread listens on this queue and sends the entries in the queues to the Alert Server. The first argument in this API is a structure that holds the topic, alert type and userid.

alt_unregister_1(reginfo, cl_handle);

This API is very much similar to the subscription API except that the request is put in a separate queue called the unregister queue.

5.3.3 Role of Java Native Interface

Since the proxy server is implemented in C and the Alert Server in JAVA, at some point it is essential to convert C structures to Java objects and invoke the remote

methods that the Alert Server provides in its interface. The proxy server is a client for the alert server; therefore it is important that requests in C and C++ be converted to java objects that can be passed as parameters to methods that are provided by the alert server to its clients for communication purpose. Java Native Interface (JNI) [17] has been used for this purpose. There are other alternative approaches that allow Java applications to interoperate with code written in other languages. For example:

- Through a TCP/IP connection or through other inter-process communication (IPC) mechanisms.
- Distributed object technologies such as Java IDL API.

A common characteristic of these alternative solutions is that Java application and native code reside in different processes. Process separation offers an important benefit. The address space protection supported by processes enables a high degree of fault isolation. But in this case, the native code of the proxy server has to convert the C structures internally and then forward it to the alert server. Therefore the native code has to communicate with Java application that resides in the same process. This is when the JNI becomes useful because

- Lesser overheads of copying and transmitting data across different processes. Loading the native library into a Java application or invoking a Java Virtual Machine from the same process is much more efficient.
- Having an application span multiple processes could result in unacceptable memory footprint. This is typically true if these processes need to reside on the same client machine. Loading the java virtual machine into the process hosting the application requires less system resources than starting a new process and loading the java virtual machine into that process.

Apart from the conversion of requests and responses from C to JAVA and vice versa, JNI is also used for making remote method calls to the Alert Server. The Alert

Server provides a remote interface, which the clients can use for communicating with the server. The reference for the remote object is obtained with the JNI functions and the methods in the interface are invoked from the C proxy server.

5.4 Initialization of Proxy Server

The proxy server needs to be initialized before it can accept calls from the clients. The initialization of proxy server basically involves three essential steps

- Set the IP address. The IP address of the machine on which the server is running is set so that the Alert Server can send the responses back to the proxy server which can latter forward them to appropriate clients.
- Create a Java Virtual Machine. This step is essential, as we need a java virtual machine in order use a Java Native Interface. The JNI invocation interface requires the linking of native programs with a Java Virtual Machine implementation. This is done by creating a JavaVMInitArgs structure in the native application and also setting the virtual machine initialization arguments stored in JavaVMOption array. The classpath and other important things are set here. After setting up the virtual machine initialization structure, the native application calls JNI_CreateJavaVM to load and initialize the Java virtual machine. The JNI_CreateJavaVM function fills in two return values:
    - An interface pointer, jvm, to the newly created Java virtual machine.
    - The JNIEnv interface pointer env for the current thread. This pointer is essential since the native code accesses JNI functions through the env pointer.
    - The last step is configuration step. In this step the “proxyconfig.txt” is read and all the macros that the application uses are set. The port number on which the alert server listens for requests and the IP address of the Alert Server is read and set. Apart from this, it also sets the proxy USER ID by reading from

the “config” file. This USER ID should be unique from all the other clients of both the Alert Server and the clients of the proxy server.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

This thesis presents a messaging system that allows separate client applications to be combined into a flexible and reliable system. It discusses the design and implementation of Alert Server that handles JAVA clients and is also responsible for distributing alerts. It explains the design and implementation of Proxy Server that handles C/C++ clients. It discusses the problems in a messaging environment and the design choices made to solve these problems. It also discusses the various alternatives available and reason behind the choice of a particular alternative.

Alert Server has been implemented using publish/subscribe model where clients can subscribe to three topics TAG, USER and PROFILE. It supports regular expression based subscription in the form of TAG. It also assures delivery of alerts on the basis of priority and maintains transaction logs for a comprehensive audit trail of delivery of alerts, acknowledgements, and receipts to appropriate destinations. It supports persistence of alerts by logging them on to a file-based storage. The clients have the option of sending non-persistent alerts also. A new logging and retrieval mechanism has been implemented. This is more efficient than the traditional way of storing the alerts. This is achieved by storing alerts on the basis of their priorities and storing additional information in the logs. A sweeping algorithm has been designed to sweep the data structures and find the consumers for the alert. A proxy server has also been designed and implemented to handle C/C++ clients. This proxy server is transparent to the clients and

its architecture supports a store and forward mechanism. The C/C++ clients are handled in the same way as JAVA clients.

6.2 FUTURE WORK

The current implementation of Alert Server is a stand-alone application. This can be integrated with GED/LED [1, 3] so that the clients have an option of either generating events or send/receive alerts. This can be achieved by changing the API of the LED and API of the Alert Server so that the client use that API to decide whether to send an alert or generate an event. The current implementation of Alert Server handles the TAG, USER and PROFILE on one server. This can be changed such that the subscriptions pertaining to USER and PROFILE can be handled by one Alert Server and those pertaining to TAG by another Alert Server. This may be done as Alert Server handles TAG subscriptions slightly differently than the other two. More over this also improves the scalability of Alert Server. The current implementation of Alert Server supports publish/subscribe model. It can be extended to support point-to-point model also. The clients in the current implementation need the IP address of the server in order to communicate. This can be changed to improve availability by using JINI [19]. The programs can then interact spontaneously enabling services to join or leave the network with ease. This allows clients to view and access available services with confidence. The current implementation can also be extended to provide administration utility and also provide encryption mechanism when the alert is sent over a network to provide security.

REFERENCES

1. Tanpisut, W., *Design and Implementation of Event based subscription/notification paradigm for distributed environments*. 2001, The University of Texas at Arlington.

2. Dasari, R., *Design and Implementation of a Local Event Detector in Java*, in *CISE*. 1999, Univ. of Florida: Gainesville.

3. Dasari, R., *Events And Rules For JAVA: Design And Implementation Of A Seamless Approach*, in *Database Systems R&D Center, CIS Department*. 1999, University of Florida: Gainesville.

4. Bloomer, J., *Power Programming with RPC*. 2000: Reilly.

5. R., R.B. *Making the Most of Middleware*. In *Data Communications International 24*. 1995.

6. Vondrak, C., *Message-Oriented Middleware*. 1997.

7. IONA, *The OrbixWeb 3.0 Programmer's Guide*. July 1997.

8. OMG, *The Common Object Request Broker: Architecture and Specification Version 2.0.*

9. Microsoft, *The Common Object Model Specification Version 0.9*. 1995.

10. Microsoft, *Distributed Component Object Model Protocol-DCOM/1.0, draft.* 1997.

11. SunMicrosystems, *Java Remote Method Invocation Specification.* 2000.

12. SunMicrosystems, *Java Message Service Specification Version 1.0.2b*. 2000.

13. IBM, *MQ Series - An Introduction to Messaging and Queuing.* 1999.

14. IBM, *MQ Series Queue Manager Clusters*. 1999.

15. FioranoMQ, *FioranoMQ and Progress Sonic MQ Comparison*. 2001.

16. Schmidt, D.C. and S. Vinoski, *The OMG Events Service. C++ Report*. 1997.

17. Liang, S., *The Java Native Interface Programmer's Guide and Specification*. 2002.

18. Bradford Nichols, D.B.J.P.F., *Pthreads Programming*. 1996.

19. SunMicrosystems, *JINI Specifications*. 2000.

BIOGRAPHICAL SKETCH

Nishanth Reddy Vontela was born on September 18, 1977 in Warangal, India. He received his Bachelor of Science degree in Mechanical Engineering from Visvesvaraya Regional College of Engineering, Nagpur, India in May 1999. In the Fall of 1999, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington in May 2002. His research interests include active databases and distributed systems.