Automated Information Distribution in Low Bandwidth Environments

by

Rabih M. Zbib

B.E. in Computer and Communications Engineering
American University of Beirut (1996)

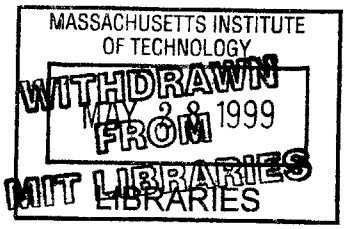
Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999



Signature of Author .....
Department of Civil and Environmental Engineering
17 May 1999

Certified by .....
Steven R. Lerman
Professor of Civil and Environmental Engineering
Thesis Supervisor

Certified by .....
V. Judson Harward
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Andrew J. Whittle
Chairman, Departmental Committee on Graduate Studies

Automated Information Distribution in Low Bandwidth Environments

by

Rabih M. Zbib

Submitted to the Department of Civil and Environmental Engineering
on 17 May 1999, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis presents a distributed trigger architecture for information distribution in distributed environments where data changes are dynamic. It supports the monitoring of trigger conditions dependent on inputs from local and non-local clients and the execution of trigger actions which may include notification or updating of persistent state. The architecture is designed as a set of independent services, which can be layered on top of multiple transport layers and employ various serialization/consistency models. A prototype implementation built on top of Java and ObjectStore's PSE Pro persistent store is also presented[1].

Thesis Supervisor: Steven R. Lerman
Title: Professor of Civil and Environmental Engineering

Thesis Supervisor: V. Judson Harward
Title: Principal Research Scientist

[1]Prepared through collaborative participation in the Advanced Telecommunications & Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0002. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not-withstanding any copyright notation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of the Army Research Laboratory or the U.S. Government.

Acknowledgment

First and foremost, I would like to thank my advisor Steven Lerman for his invaluable support since I first came to MIT in the Summer of 1996 as a visiting student and throughout my stay later as a graduate student. I would also like to thank him for his great feedback on my thesis.

I am equally thankful to Judson Harward, my research supervisor, for his remarkable insight which contributed greatly to the ideas presented in this thesis, as well as his continuous support. Dr. Harward's insistence that I write more clearly contributed to the readability of this thesis.

I thank my colleagues who worked on the ATIRP project: Saadeddine Mneimneh, Julio Lopez and Richard Rabbat. I also thank the staff at CECI for the very special work atmosphere they have created through the years.

To the many people who were more of a second family than they were friends, thank you for making life at MIT possible, and even sometimes enjoyable. Among them I would like to mention Issam, Saad, Hisham, Nadine, Fadi, Richard, Assef, Mahdi, Walid, Maysa, Louay, Yehia, Shankar, and especially Mary.

To my friends abroad, especially Ali, Amal, Darine, Fadi Ibrahim, Fadi Zaraket, Marwan, Valia, and Wassim, even though I did not keep up with your pace of emails, you were always on my mind.

To my parents, whose children's education remained their biggest concern through the darkest and most difficult years, I ought it all.

To Dad, Mom, Ihab, Malak and Youssef...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The rise of the Internet opened the way to a whole new paradigm of software applications. In typical internet applications, one host is interested in the information available on another (e.g. the World Wide Web). Currently, most applications of this type follow the client-server model, where the server process constantly listens to requests initiated by clients. The initiative for communication in this model is on the client's side, since the client pulls information by initiating requests to the server. Such a model is convenient in cases when the data on the server is static, or when it rarely changes. However, when the server contains information about a dynamic environment that changes unpredictably, client polls become an insufficient way for the client to obtain a replica of the server's data.

The solution to this problem might look simple: If host A has the dynamic data, and host B is interested in it, then B subscribes with A, and runs a server process. Whenever A has new information it sends it to B by talking to B's server process. This simple solution has many problems however. The first problem is that almost always the network does not have enough bandwidth to transmit all the information that changes on A. The problem of bandwidth is the bottleneck in real-time internet applications today, and it is likely to remain a problem in the foreseeable future. No matter how much improvement is achieved on the level of the network physical layer, new applications will require more bandwidth. In addition, clients may not be interested in all the information available on the server hosts; therefore, a mechanism

for data filtering is required. Finally, in some applications hosts are susceptible to crashes. A persistence scheme is needed to enable applications on both sides of the network connection to recover.

This thesis describes a light weight, distributed trigger architecture that attempts to solve this problem, by including aspects from active databases (see Section 2.1), and publish-and-subscribe systems (see Section 2.2).

1.2 Applications

This section describes applications for the architecture presented in this thesis. First, I describe the problem of information distribution among low echelon military units, which is the main motivation for this research. Then I discuss other potential applications for the architecture.

1.2.1 Model-Based Battlefield

The distribution of information is essential to the command and control of the battlefield on all levels. The traditional mechanism of information distribution was based on the exchange of voice messages. The significant reduction in computers' size, cost, and power consumption has made it possible to install computing systems on military units at all echelon levels. The mechanism for information distribution shifted from the exchange of voice messages to the exchange of text messages. In this paradigm, text still simulates voice, since messages are intended for human comprehension, and computer-to-computer communication only serves as a replacement for voice communication (see Figure 1.2).

This approach has several drawbacks. First, the number of message types needed in an environment such as the battlefield is very large. Message comprehension then becomes a problem. Second, the number of messages that needs to be exchanged is also very large; which puts a burden on human personnel and on the underlaying communication network. Finally, human-oriented messages do not take advantage of the computer's powerful capabilities of data abstraction. Such abstraction facilitates the standardization of battlefield modeling (see [Cha95]).

Chamberlain ([Cha95]) notes that there is a major difference between battlefield modeling

and simulation, on the one hand, and the way battle command is actually done, on the other. He advocates that these two tasks can be carried out similarly, through the concept of *model-based battlefield command.* In a model-based battlefield, different units are viewed as each possessing their own versions of an abstract model of the real battlefield environment. Different units update each other's models through the direct exchange of messages that are not necessarily human comprehensible. Human users interact with the model through external applications that translate between the model's abstract data and human comprehensible data forms.

Among the advantages of this approach that Chamberlain cites is that it facilitates the automation of tasks. For instance, the model need not only interact with human-ended applications. It can read data from sensors, a GPS system, etc. Another advantage is that the model-based battlefield facilitates the abstraction of information. The model can use complicated data structures, and define hierarchies of data structures that map the real environment better than text messages.

A third important advantage of the model-based battlefield representation is a reduction in the amount of necessary communication. Since each unit keeps its own version of the model, the system can be viewed as a replication mechanism. Messages are only exchanged to enhance the consistency of the model as information changes. The reduction in the amount of communication is essential since the computation power present on units (especially at the lower echelons level) far exceeds the bandwidth of the communication networks that connect them.

Databases are a good tool for mapping a model such as the battlefield. They can store large amounts of data, and they have the ability to recover from crashes, which are likely on the battlefield. Active database triggers, or alternatively ECA rules (see Section 2.1.3), are used in this context as the automation tool for information distribution. They describe criteria for model updates, and thus, together with network delays, determine how inconsistent a model can be. One extreme is reached when the bandwidth of the communication network can accommodate all the traffic necessary for full consistency. The other holds when the network fails completely; in this case, the models can only rely on prediction by the model for its update. The role of triggers falls between these two extremes.

Network conditions are very dynamic on the low echelon level of the battlefield, where the physical layer is usually wireless. [Cha95] suggests that triggers should be adaptive to

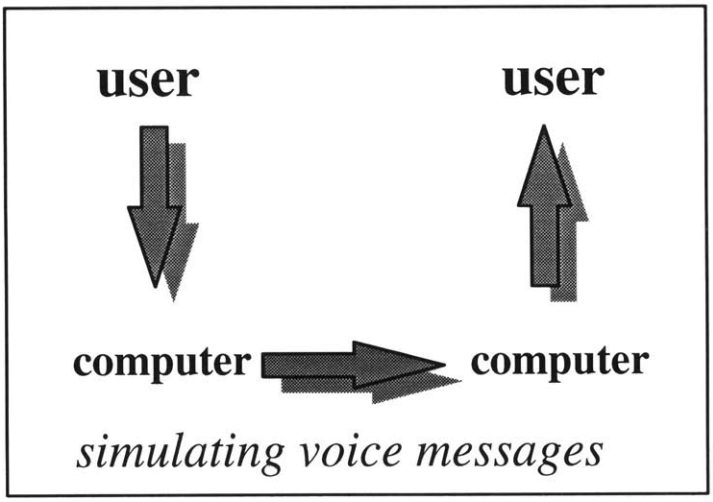


Figure 1-1: Message based battlefield

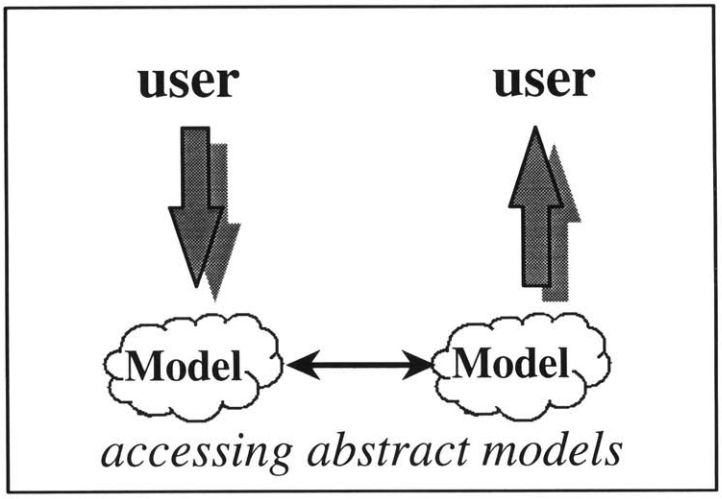


Figure 1-2: Model based battlefield

these network conditions. More precisely, update criteria should depend on the condition of the network. Statistics collected by nodes are a means of determining this condition. The architecture presented in this thesis does not provide such functionality; however, it presents principles of design that can be extended in the future to accommodate adaptive triggers.

1.2.2 Other Applications

As we mentioned before, the motivation for the architecture described in this thesis is the automation of information distribution on the battlefield, especially at the lower echelon level. However, there are many other situations where the principles of the architecture are appropriate. These situations can be described in general terms. When the system is large, and data owned by different nodes change very dynamically, it is impractical to implement traditional distributed database concurrency algorithms that use serialization. Trigger architectures, in this case, provide better performance.

In recent years, some of the research in the area of air traffic management systems has been concentrated on the replacement of traditional centralized, ground-based management by a decentralized model where aircraft can plan their trajectories in real-time (see [PT+97]). This approach is now possible thanks to GPS systems and powerful computers installed on aircraft. This problem can be modeled as a set of nodes (aircraft), where each has its own view of the environment (e.g. position, weather information, its own trajectory, other aircraft trajectories, etc.). The different views update each other according to criteria that can be defined using triggers, to increase each other's situation awareness. The architecture in this thesis does not directly address the free flight air traffic control problem; however, the ideas and design principles described here are applicable.

The stock market represents another similar situation. Even though the bandwidth constraint is not as severe as it is in the previous cases, stock trading software requires alerts and information filtering since the amount of information available is huge. The principle of monitoring and notification to subscribers when certain conditions are met is required in this case. Distributed triggers can implement monitors and notify subscribers when preset conditions are met.

1.3 Thesis Summary

The rest of this thesis is organized as follows:

In Chapter 2 discusses the previous work done on active databases. I describes the development that led to the ECA rule abstraction, and discuss the common elements of active databases that the community agrees on. Chapter 2 also includes a discussion of the Internet publish and subscribe technology.

Chapter 3 starts by defining the functional requirements of the architecture clearly. Then it discusses the shortcomings of existing technologies in satisfying these requirements. Next, the advantages of the unbundled services design are discussed. A high level description of the architecture follows, including the network model and the data model assumed by the architecture. The remaining sections of this chapter describe the different services and their interaction in detail.

Chapter 4describes the Java prototype of the architecture, and discusses issues that arise in the implementation.

Chapter 5 concludes the thesis by summarizing its content and describing possible future work.

Chapter 2

Previous Work

The first section of this chapter discusses active databases: the motivation behind them, the general concepts as defined in the literature, and the different approaches that active database systems take in implementing these concepts.

The second section describes a general distributed application model known as publish and subscribe. Unlike active databases, the general concepts in publish and subscribe technologies are not as formally defined. The discussion in that section is, therefore, mainly presented through examples.

2.1 Active Databases

Active Database Management Systems (ADBMS) are an integration of traditional Database Management Systems (DBMS) and AI expert systems. They incorporate the transactional and querying capabilities of traditional databases with expert systems' ability to maintain rules and respond to events.

In their most general form, active databases extend traditional databases by allowing for the installation of rules that monitor events, check conditions and take actions when the conditions are met (hence the name Event-Condition-Action, or ECA, rules). The incorporation of database capabilities into rule systems is important when the data's size makes it impractical to store in main memory, or when the application is failure-prone, and the data needs to be made persistent. In addition, DBMS allow for transactional operations and queries.

From another point of view, active databases have many advantages over conventional databases. They can perform some operations more efficiently. For instance, suppose some action needs to be taken when a certain state in the database occurs. In the case of conventional databases, an application needs to constantly check the state of the database, while an active database can efficiently implement this behavior through a rule. [WC96] mentions other functionality of active databases that are beyond the scope of passive databases. For example, passive databases allow for simple integrity constraints such as unique keys or referential integrity constraints (i.e. if one data item references a second item, referential integrity guarantees that the second actually exists). Active databases can implement more general and complex integrity constraints as ECA rules. Active Databases perform well when communication between the database and external applications is required after the execution of certain database operations.

Active databases have applications in such diverse areas as network management, medical applications, coordination of distributed systems and air traffic control.

2.1.1 Historical Background

[Day95] and [WC96] both trace the roots of active databases back to CODASYL in the early 70's (see [COD73]). CODASYL specified a Data Description Language (DDL) that allows the declaration of user-defined record types and the specification of database procedures to be executed when specified operations such as INSERT, REMOVE, STORE, DELETE, MODIFY, FIND and GET are performed on the record. Database procedures in the CODASYL system can be implemented in general purpose programming languages as long as they adhere to certain rules. Other early examples of database systems that included rules to maintain consistency constraints are IBM Query-by-Example ([Zlo77]) and IMS/VS([IBM78]).

The term a*ctive database* was first introduced in [Mor83]. The first clear abstraction of ECA rules, however, was presented in the HiPAC (High Performance ACtive database system) project in the mid-80's ([MD89]).

2.1.2 Data Model

As we mentioned in the beginning of this section, active databases were conceived as an extension of traditional DBMS systems, and they are in some cases built on top of them. Most active database projects assume an underlying data model that is either relational (e.g. AIS, Ariel, Postgres, Starburst) or object-oriented (e.g. Chimera, HiPAC, Ode, SAMOS, Sentinel, ACOOD, REACH).

The degree of coupling between the rule system and the underlying database varies between different ADBMS's.

2.1.3 ECA Rules

Event-Condition-Action (ECA) rules represent the most common abstraction of active databases. The desired active behavior of the system is specified through these rules. In general, the *event* is an instance of a database operation, the *condition* is a boolean function of the state of the database evaluated when the event happens, and the *action* is a database operation or external procedure that is executed (referred to as firing) when the condition is true. ECA rules are also referred to as triggers. Variations of these definitions have appeared in the different research projects in the area of active databases (see [Day95] , [WC96] and [PD+93]).

The functionality of Active Databases is largely determined by their *rule language*, which specifies the semantics of events, conditions and actions. The rule language determines the capabilities, complexity and execution constraints of the system.

Events

Events describe situations in which the system may react, by causing specific rules to be checked. Rule languages define *event types,* which are generic. Event occurrences are instances of event types. Events can be either primitive or composite. The most common types of primitive events are operations on the database, such as data modification (create, update, delete), and data retrieval. Temporal events allow for the specification of rules that express temporal logic. Examples of temporal events are reaction to absolute time, at periodic intervals, or operations such as *before* and *after.* Another type of primitive event is external events; they do not include

database operations, but cause rules to be checked when they are signaled to the database by applications.

Composite events are constructed by combining primitive events in different ways through logical or temporal operators. Composite events can also rely on sequences, or more generally on a history of primitive events. Some rule languages restrict primitive events that construct a composite event to belong to the same database transaction, while others allow primitive events from different transactions to be combined.

Some rule languages define parametrized events, where the event passes parameters to the condition and action. In this case, the system needs to provide a mechanism for passing parameters from events to conditions and actions.

Conditions

The database must be in a specific state when an event of an ECA rule happens in order to execute the action. Rule conditions check this desired state to determine whether the rule should fire. In relational ADBMS's, conditions are typically database predicates (such as the SQL *where* clause), or database queries, where the condition evaluates to true if the query is not empty. In the case of object oriented ADBMS's, conditions are queries or procedures that access database objects.

The separation of events from conditions allows for the system to react to events other than database operations; it also allows more flexibility in rule specification, since different actions might need to be taken when the condition is satisfied, depending on the event. Seperating events also allows a more efficient execution of rules; conditions are typically queries that are expensive to evaluate. The condition is only evaluated when the event indicates that its value may have changed.

Action

The ADBMS executes an ECA rule whenever an event occurs that causes the rule's condition to evaluate to true. An action can be a transaction on the database itself, written in the underlying DBMS's Data Manipulation Language (DML). In addition to normal database operations, an action can be a rollback operation when the rule is intended for recovery from transactions that

violate integrity constraints. An action can also be an external procedure in an application, written in a general purpose language. Some ADBMS's (such as SAMOS [Gat96]) allow actions to be user notifications.

Some rule languages allow rules to specify additional properties such as rule priorities, and grouping rules into structures. They also define commands for rule management; semantics of creation and deletion are defined by the rule language.

2.1.4 Execution Model

[ACM96] specifies that an ADBMS should define an ECA rule execution model. A rule execution model defines the behavior of rules, and their interaction with database operations, given the semantics of the rule language. In particular, it defines the granularity of rule execution, that is the level of operation at which rules are evaluated (e.g. database command, transaction, session, etc.). Granularity directly affects the results of rule execution. [Day95] gives examples of the effect of granularity on rule execution results. An execution model also defines how rules interact with each other. Some systems execute rules sequentially, while others allow for concurrent execution. It can also specify algorithms for conflict resolution when several rules are triggered simultaneously due to the same event.

Since both events and actions usually consist of database operations, both are executed within transactions. The relationship between the transaction that causes a rule to trigger and the resulting transaction is referred to as the coupling mode. The most common modes of coupling are:

- **immediate:** The triggered transaction is executed directly after the triggering event.
- **deferred:** The triggered transaction is executed after the triggering transaction is finished, but before it is committed.
- **decoupled:** The action is executed in a completely different transaction.

The first two modes are usually simpler to implement, but they result in longer transactions. The decoupled mode, on the other hand, results in shorter transactions. It is also more flexible, since a triggered transaction can commit or abort independently of the original transaction.

2.1.5 Implementation Issues

The rule language and the execution model define a theoretical model of the system; however, many issues arise when the implementation of such a system is attempted.

The system can be either built on top of an already existing passive database system (e.g. HiPAC [AG89]), or it can define its own persistence system. For example Ode [MD89] define a persistent object oriented language similar to C++ called O++. An active database needs to implement a rule management user interface, as well as a behavior in face of syntactic errors in rule definitions and runtime errors in rule execution. The system should also implement a mechanism for the detection of events. Such a mechanism is not obvious in the case of temporal and composite events. The efficiency of condition evaluation becomes another issue when the rule base grows large. A recovery procedure is needed in case of crashes. Depending on the coupling mode of the system, the recovery mechanism may be more complicated than that of passive DBMS's (e.g. decoupled systems need to worry about non-executed action transactions).

2.2 Publish and Subscribe Technology

2.2.1 Client-Server Computing

The client-server model is an abstraction that defines interaction between processes that use a peer-to-peer communication protocol, such as TCP/IP. It solves the problem of rendezvous that arises when two symmetric processes try to initiate a communication ([CS96]). In general, the application that initiates the communication is called the *client*, while the *server* is the application that responds to the requests of clients by listening to them at a certain known address (IP address + TCP port in the case of TCP/IP). The server serves the client's request by performing requested calculations, if any, and returning the result to the client.

The introduction of personal computers and the advancement of computer networking in the 1970's and 1980's led to the replacement of time-shared mainframes by the client-server model as the paradigm for multiuser systems. The popularity of the model increased with the evolution of the internet. While a mainframe is a multiuser environment with centralized processing and distributed dummy terminals, the processing in the client-server model is decentralized. The

advantage of this model is that it reduces the cost of computing. It also offers more scalability, allows for duplication, and with protocols such as TCP/IP it allows interoperation of different platforms on different types of networks.

2.2.2 Server Push

The client-server model described in the previous section puts the initiative on the client's side; the server only reacts to client requests. The asymmetry between the client and the server solves the problem of communication initialization. In the case where the server contains static data, this scheme works well. However, when the server maintains data that changes dynamically, the client needs to poll the server by submitting regular requests in order to stay synchronized with the server's state. Periodic polling is not enough if the changes on the server are unpredictable.

To solve this problem, the server needs to push information to interested clients whenever its state changes in a manner relevant to them. A fact worth noting is that this functionality is described on a higher level of abstraction than the traditional client-server model. A client as described here might need to run a TCP server socket in order to listen for the server pushes. Therefore, in the subsequent discussion we assume that the "client" may maintain its own server socket to allow it to listen to the original server.

2.2.3 Internet Push

Recently, many internet push applications appeared as extensions to static web browsers. Products like PointCast, Marimba's Castanet and Netscape's Constellation allow users to subscribe to certain information channels. Updates on these channels are pushed to the browser application at specified intervals. Typically, the subscription to different channels is made through customizing some kind of a profile on the browser's side. It is still unclear to which extent this kind of usage of the Internet will replace the static web browsing. While it might look like a solution to the problem of finding relevant data on the web, some argue that its passivity limits the choice of web users.

2.2.4 The Object Management Group (OMG) Event Service

CORBA

The Common Object Request Broker Architecture (CORBA) is an abstract architecture for distributed objects defined by the Object Management Group (OMG) consortium (see [OMG91]). CORBA provides a common, object oriented framework for writing distributed applications. Objects in CORBA provide services to remote applications through requests, which are in the form of remote method calls on the service objects. CORBA requests are handled by Object Request Brokers (ORB's) which are responsible for finding the object implementation and communicating the data between the requester and server. The location and implementation of objects is, therefore, transparent to the requesting clients who only deal with object interfaces. Objects implemented in different programming languages on different platforms can exist on the same ORB. Object interfaces are specified in the Interface Definition Language (IDL), which provides the clients with signatures of the service objects.

CORBA implementations are available in many programming languages on different platforms.

The Event Service

In the CORBA model, a client needs to be aware of the existence of the service object and of its method signature. The OMG Event Service ([OMG97]) defines a more decoupled computational model, built on top of CORBA, where event data generated by object operations (suppliers) are communicated to interested parties (consumers).

The communication between suppliers and consumers is made through a CORBA object called an event channel. By allowing multiple consumers and suppliers to communicate asynchronously, the event channel can be viewed as a type of subscription service. The OMG Event Service specifies two models of communication between suppliers or consumers and event channels.

1. The push model: The supplier pushes the event data to the channel, which in turn pushes it to the consumer. The initiative is, therefore, on the side of the supplier.

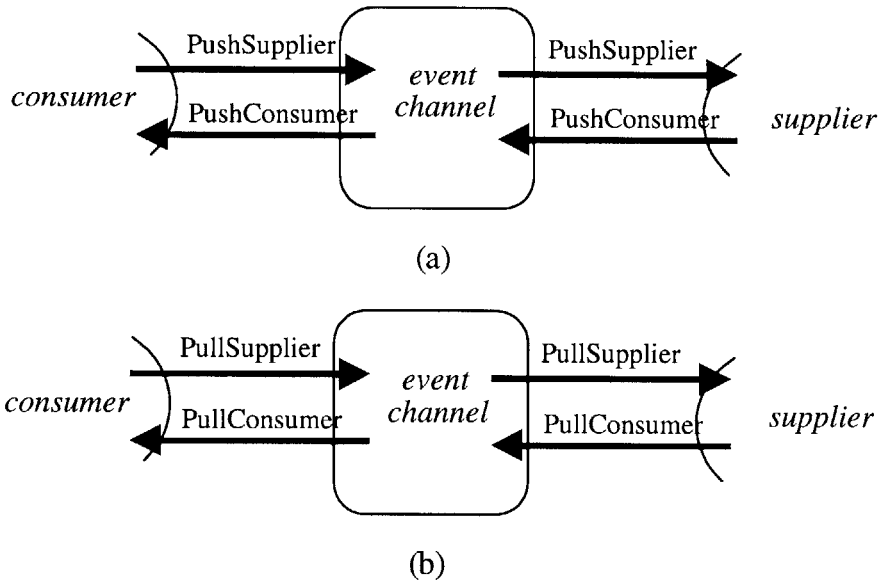


Figure 2-1: The CORBA Event Specification: communication between a Supplier and a Consumer through an Event Channel in (a) Push style and (b) Pull style

2. The pull model: The consumer pulls the event data from the channel, which in turn pulls it from the supplier. The initiative in this case is on the side of the consumer.

Since event channels are not synchronous, in the sense that event data on an event channel need not be communicated instantly, a mixed style communication can be performed. The supplier can push event data to an event channel, and the consumer can pull the data from the same channel.

As in the case of CORBA, OMG only defines abstract specifications of the supplier, consumer and event channel. The specifications are written as abstract (similar to C++ virtual) classes in IDL. Commercial products provide implementations for these specifications in different programming languages, mainly C++ and Java.

Chapter 3

Architecture

This Chapter presents a trigger architecture for automated information distribution. The first section specifies the functional goals of the architecture, given certain assumptions about the environment. Section 3.2 explains why the technologies presented in Chapter 2 do not provide a ready-to-use solution for this problem.

Section 3.3 gives a high level view of the distributed components in the system, and briefly explains their interaction. Section 3.4 introduces the data model that the architecture assumes, together with a rationale for the model.

Section 3.5 argues for the design of the architecture as a set of services. The subsequent sections describe the services that constitute the architecture. This chapter concludes by introducing the concept of abstract higher level services that are built on top of the architecture services.

3.1 Functional Goals

Chapter 1 argued for the importance of the automation of information distribution in dynamic, distributed environments. This section states the functional goals for a system that attempts to solve this problem.

3.1.1 Data Filtering

In an environment such as the battlefield or the stock market, the amount of information available is overwhelming, which makes it impractical for humans to track. The system should, therefore, provide a mechanism to filter important data and present it to users. An additional argument for data filtering is that each user is usually interested in a specific subset of data items from those available in the environment. The architecture attempts to free the end user from the burden of selecting interesting data. Users should be able to express their interest through subscriptions. The system notifies users only about updates to data that they have subscribed to.

Subscriptions to specific data filters the environment in the space of data variables. However, since data usually changes frequently, filtering in the temporal dimension is also needed. In other words, it is often impractical to notify interested users about all changes of data. This becomes a bigger problem when the bandwidth of the underlying network is limited. Users, therefore, should be able to express criteria for updates that depend on the temporal change of data or on other properties of its values. Trigger conditions are an appropriate implementation of these criteria. Moreover, triggers allow users to express more complex criteria which involve more than one data variable. For example, when a user needs to be notified when variable x exceeds variable y by a certain threshold δ, the user should be able to subscribe to a trigger with such a condition instead of subscribing to changes on both x and y.

This example illustrates another advantage of data filtering besides reducing the burden on end users; it reduces the amount of information that must be communicated for model updates. In the case where x and y belong to the same host, only changes that affect the relevant condition need to be transmitted. These are fewer than changes to the primitive data variables x and y.

3.1.2 Increased Availability

The main goal of replication systems in distributed environments is to increase the availability of remote data to local users by decreasing the time needed to access it. Data should be available to users at their local hosts. A simple replication scheme can be used when the data is static. However, when data changes frequently, the system should attempt to minimize the

amount of time when the data replica are obsolete. Triggers in this sense attempt to increase the consistency of these replica by updating the different versions of the model. Trigger conditions are criteria that describe situations when the replicated data become obsolete, and therefore need to be updated. They propagate important updates between different hosts to increase the utility of the replicated data.

3.1.3 Alerts

In any automated system, humans are always the ultimate users. When critical situations arise, especially in the case of the battlefield, human intervention is necessary. Again, triggers can be viewed as monitors. End users are alerted when critical situations occur.

3.1.4 Persistence

In the model-based paradigm, each host maintains a certain version of the abstract environment model. In the case where hosts are failure prone (such as the battlefield), the model needs to be persistent to permit recovery. The persistent state of the model will include replicated data, but more important, it will include triggers and subscriptions that were installed before crashing. The recovery of triggers and subscriptions allows the host to resume its contribution in the ongoing model update process after recovery.

3.2 Shortcomings of pre-existing technologies

The fields of active databases and publish and subscribe technologies (summarized in Chapter 2) each can partially achieve the functional requirements listed above. This section, however, will argue that the available technologies in both of these fields are neither sufficient by themselves nor can be simply integrated to solve the specific problem presented here.

3.2.1 Active Databases

Active databases have the advantage of implementing triggers, which are an appropriate mechanism for expressing criteria to push information. Active databases, however, are intended to satisfy different requirements than those stated in Section 2.1. First, active databases, es-

pecially in a distributed setting, aim at achieving complete consistency, a goal that is too expensive in terms of available bandwidth in the environments for which this architecture is intended. Among other things, distributed transactions, where more than one host contribute to the same transaction before it commits, require the hosts to exchange a large number of messages to ensure that the transaction appears atomic to users (see [LM+94]). In the case of this architecture, however, the purpose of triggers is to duplicate information, and thus achieve partial consistency. Moreover, triggers guarantee that this partial consistency is bounded, where bounds can be implied from trigger conditions and network connectivity.

As it will become clearer in the subsequent sections of this chapter, user notification is an essential part of trigger actions (in the sense of ECA rule action). Active databases, in general, couple the rule management modules (activity modules) with the underlying DBMS. This coupling does not allow the thread of control to be transformed from the main database thread to the notification mechanism. As will be shown in Section 3.8, automated installation of remote triggers is necessary in some cases. This automation turns out to require a mechanism similar to that of subscriptions, and thus cannot be implemented in a natural manner using tightly-coupled active databases.

3.2.2 Publish and Subscribe

In contrast to active databases, publish and subscribe technologies are a good infrastructure for information duplication, especially since in these systems, unlike web browser caching mechanisms, the information is pushed from the server side. The data owner, therefore, can be viewed as the server, and subscribers are the owners of data replica (see [BB+98]). The disadvantage, however, is that publish and subscribe systems, in general, do not accommodate complicated criteria for pushing information. In particular, as one can deduce from the functional requirements listed above, these criteria should include properties of the data itself, a requirement that trigger conditions can implement. The channel abstraction of the OMG Event Service is an appropriate abstraction for the distribution of specific data items to specific clients. A generalization of this abstraction is to view triggers as information distribution channels (see Section 1.10).

Finally, publish and subscribe technologies do not have persistence capabilities that fit the

requirements of environments such as the battlefield.

3.3 Network Model

I start the description of the trigger architecture by giving a high level description of the different components involved. The architecture assumes several *hosts* (*nodes*) connected by a low bandwidth network. A node represents a physical host. Each of the nodes possesses a version (*view*) of the model of the environment. In addition, each node runs a number of processes called *clients*. Clients use the services provided by the system. They update their local hosts, they subscribe to triggers on their local hosts, or they may access it to read data. The connection between clients and their local hosts, in general, does not use the network (see Fig 2.1); therefore, there is no cost for the communication between clients and their local hosts. Communication between nodes, on the other hand, is carried through the low bandwidth network, and is therefore costly. It is this communication, however, that allows different nodes to update each other's views, and thus maintain partial consistency between the different views on the nodes.

For example, a node could be a host running on a vehicle in the battlefield. Many clients can run on the same host. A GPS system, for example, can update the model, while a Graphic User Interface subscribes to updates on the model such as the host's own position and the positions of other vehicles, etc.

3.4 Data Model

The data model that a database system implements influences the functionality of the system. This is true in the case of traditional DBMS's, and also in active DBMS's. A relational model, for instance, allows for efficient implementation of complex queries. On the other hand, object-oriented databases model the real environment better than relational databases, at the expense of reduced querying capabilities.

In the environments that this architecture is intended to model (e.g. low echelon battlefield), it is not likely that complex queries are needed. Efficient modeling of the large environment is, however, a more important issue. The environment is mapped to a hierarchy of objects. In the

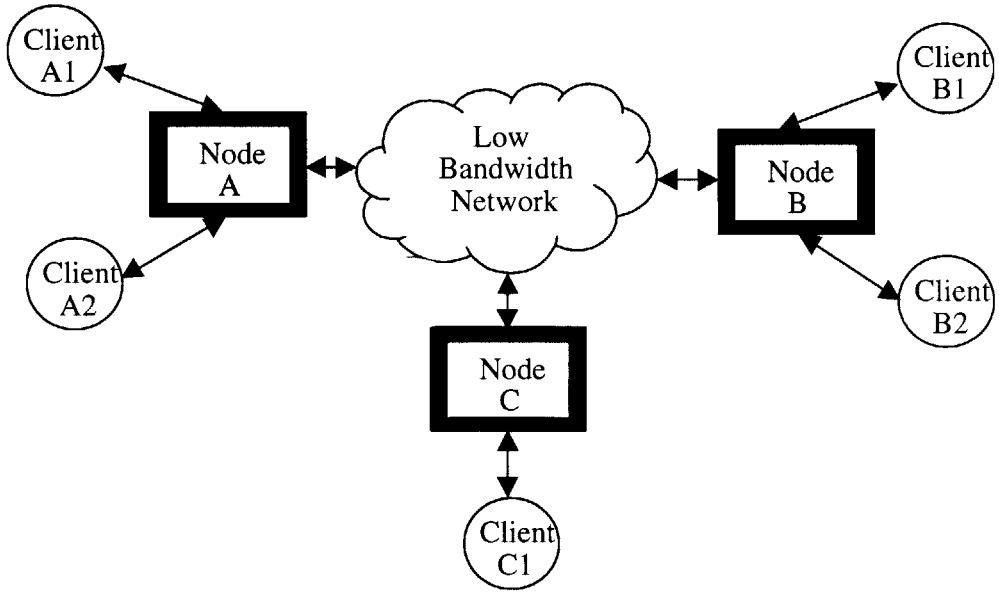


Figure 3-1: High Level System Architecture.

vehicle object of the previous example, a vehicle object could have a position object, which in turn has an x component and a y component. Each object is assumed to be owned by a single node. The highest level in the hierarchy is therefore the node. The other levels of the hierarchy map the environment in a logical and organized manner. Only local clients on the owner node can directly change data. Remote nodes can change local objects indirectly. For example, if the vehicle object in the previous example has a member that stores the difference between its node's position and that of another, a position update from the remote node changes the member that holds the difference.

The object hierarchy is mapped to a hierarchy of names. Each object is identified by a unique hierarchical name, which begins with the name of its owner node. Node names can use a network hierarchical naming system such as the TCP/IP's Domain Name System (DNS). The convention for naming is that the node name is separated from the subsequent objects' names by the '/' delimiter. Names of objects in the different levels of the hierarchy are separated by dots. For example, the following is a valid name:

$$\overbrace{ceci.mit.edu}^{node}/\ \overbrace{object1.object2.object3}^{object}$$

As will be shown in Chapter 4, names are used in trigger construction, transactions and subscriptions. A replica of an object on a remote host possesses the same name as the original object on its owner node.

3.5 Service Design

The traditional approach in active databases tightly couples the active functionality (responsible for trigger management) to the underlying database system. The system in this case is a large monolithic module. Even systems that have a layered design bind the active part to the DBMS. [GK+98] and [KG+98] state many disadvantages of this design approach. First, porting the same active modules to different DBMS's is impossible, since the active modules are built around a specific DBMS. Second, this design approach lacks a flexibility that is desired in some situations: the active components cannot be used without the underlying DBMS. More importantly, the coupled approach does not allow components to be added, or reconfigured in order to suite specific needs.

[GK+98] and [KG+98] propose another approach for active database design, where the active functionality is *unbundled* from the DBMS, i.e., the active module of the system is seen as a separate service. The system is then decomposed into a number of more-or-less independent services. Services can be used in different architectures, which reconfigure them appropriately depending on the required functionality.

The architecture presented in this thesis is designed as a set of services that interact with each other. The architecture, therefore, does not use an embedded database system, but rather a persistence service that is only used to store the node's version of the model.

The architecture is primarily composed of four services:

- **Transaction Service**: Executes transactions submitted from the local client, and from remote nodes. It also detects external data events and signals them to the trigger service.

- **Trigger Service**: Maintains a base of triggers that are installed on the node. It is also responsible for detecting internal primitive events, and composite events.
- **Subscription Service**: Manages subscription from local clients and remote nodes, and handles notifications to subscribers.
- **Persistence Service**: Provide persistence to the above three services, which need to store data structures appropriate for recovery.

The approach of service unbundled allows the database to be used as a persistence service, while another services, the transaction service, provides transactional functionality that can implement the loose, yet bounded consistency the was mentioned in Section 3.2.

The following sections describe these services in more detail. I also describe their interaction within the same node, and the interactions among different nodes. As will become clearer in the subsequent sections of this chapter, building the architecture as a set of unbundled services is an appropriate approach to achieve the functional goals set in Section 3.1, while overcoming the difficulties that active databases and publish and subscribe systems present.

3.6 Transaction Service

All operations that access the persistent model of the system database must be carried out in a local transaction, in order for the model to remain in a consistent state. The *transaction service* provides an interface to the clients for performing operations on the persistent model by submitting transactions.

3.6.1 Transactions

A transaction is a batch of operations that a client constructs off-line and submits to the system. As mentioned in the previous section, operations in transactions refer to objects by their names. Table 3.1 shows the different types of operations that can be included in a transaction. A transaction can contain as many operations as desired. A persistent object, however, must satisfy certain conditions in order for operations to be performed on it. For example, a *create*

Operation	Description	Arguments
create	Create a new persistent object.	*(object,objectName)*
update	Update the value of a persistent object	*(object,objectName)*
updateWithEvent	Update the value of a persistent object, and signal the update as an event.	*(object,objectName)*
read	Read the value of a persistent object, and return it.	*(objectName)*
invoke	Invoke a method of a persistent object	*(objectName, methodName, inputObject[], outputObject)*
event	Generate an event related to a presistent object without updating it.	*(objectName)*
destroy	Remove a presistent object.	*(objectName)*

Table 3.1: Desrciption of Transaction Operations.

cannot be performed on an object that already exists, and an *update* or a *destroy* cannot be performed on an object that has not been created previously.

Clients compose transactions off-line, and then submit them to the transaction service. The transaction service stores transactions in a persistent queue that is called the *external transaction queue.* A client transaction can be either of two types:

- **blocking:** The client submits the transaction, and blocks until it is executed, and the results are returned.
- **non-blocking:** The client submits the transaction and its thread of control returns when the transaction is stored in the external transaction queue, but before it executes.

Transactions in the external transaction queue are executed serially in their order of arrival. A transaction is atomic: either all operations in the transaction are performed or all are aborted. The effect of partial results of a transaction cannot appear in the persistent state of a model. The execution of a transaction might result in the generation of a *data event.* A data event is generated by the execution of an *updateWithEvent*, or an *event* operation (see Table 3.1). When a transaction is successful it commits, then all data events that result from the transaction are signaled to the *trigger service.* A data event causes the trigger manager to evaluate triggers that depend on that event. A trigger that fires as the result of a data event can contain an action in the form of a transaction. Such transactions are called *internal transactions*, and they are stored in the *internal transaction queue.* The transaction service gives priority

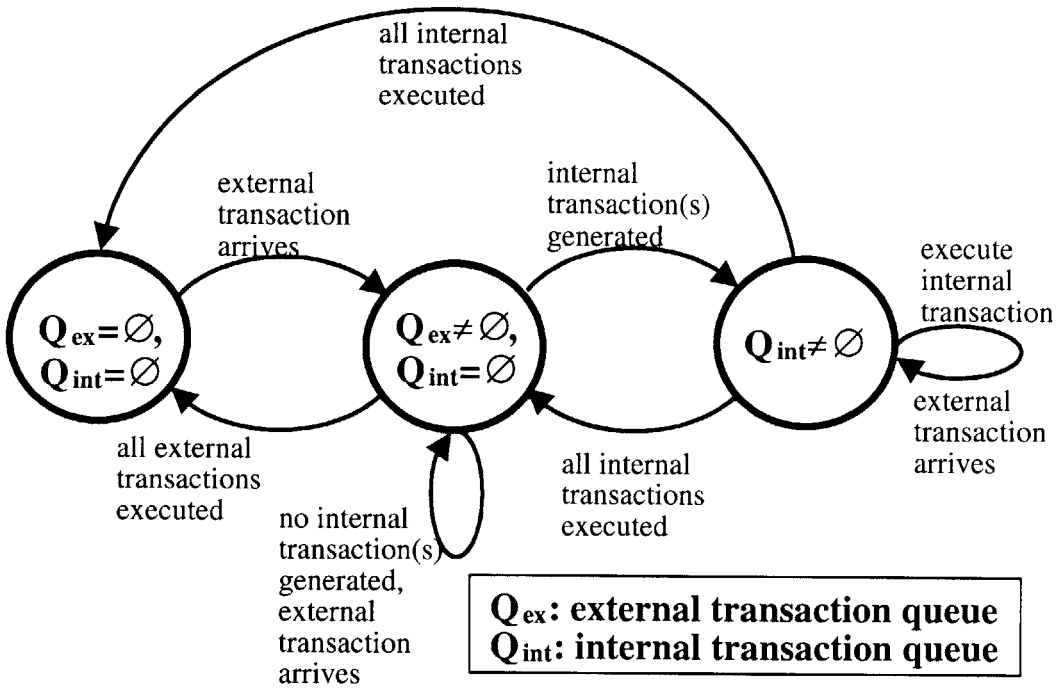


Figure 3-2: Transaction Execution Model.

to internal transactions. All internal transactions generated by an external transaction are executed before other external transactions (see Figure 3-2). In the case where an internal transaction generates new transactions, the new transactions are executed right after their generating transaction. Cascaded transactions are, therefore, executed in depth-first order. To achieve this serialization model, internal transactions that result from trigger actions are inserted at the head of the internal transaction queue. The rational behind this execution model is that all consequences of a transaction take place before any other transaction is executed. In the terminology of active database execution models (see Section 1.1.4), this execution model can be considered as decoupled, since triggers actions are executed in separate transactions. The shorter transactions that result from such an execution model are an advantage in the presence of a frequent possibility of failure.

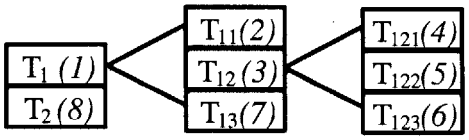


Figure 3-3: Example of Cascaded Transactions. The execution order of each transaction is shown between parentheses.

3.7 Trigger Service

3.7.1 Triggers

Before discussing the trigger service functionality, I give a brief description of triggers.

A trigger depends on a set of data variables that belong to the environment, called the trigger inputs. A *trigger event* is a data event that occurs on an input variable of the trigger. Since each data variable is identified by a unique name, an event is sufficiently represented by the name of the event. Composite events can be expressed through the *history* of the trigger.

In some cases simple replication of the trigger inputs is not sufficient to evaluate the trigger's condition. In these cases, previous values of some of the trigger inputs, in addition to other state variables are needed in order to evaluate the condition. The set of variables that is needed to evaluate the trigger condition is called the *trigger history.* For example, a temporal trigger that fires whenever variable x changes by more than δ in a time interval less that δt, needs all the updates of x that are less than δt old in order to evaluate its condition correctly. The trigger history can also maintain state variables in order to express composite events. For example, consider a trigger that fires whenever a vehicle either enters a certain region, or leaves it. This trigger needs to maintain a state variable that indicates whether the vehicle was last inside the region or outside it. The trigger history is regarded as part of the trigger, and not of the environment. This simplifies the design of triggers, and the update of the history since for each trigger one can define variables (as well as their temporal and logical characteristics) that need to be stored in the history in order to correctly evaluate the condition. In the previous example of the temporal trigger, all updates that are older than δt need not be included in the history, since they will not be used to evaluate the condition. The trigger history, therefore, must be maintained on every trigger event regardless of whether this event fires the corresponding

trigger.

Keeping the history as part of the trigger has another advantage: it keeps the model of the environment "clean" by not contaminating it with state variables, and thus allows the database to map the real environment in a better manner.

In light of the above discussion, the trigger condition is a boolean function of the state of the trigger (inputs + history) at a certain time. The condition is only evaluated when a data event on an input variable occurs.

In the terminology of active databases, an action is a user-defined procedure that is executed when the trigger condition evaluates to true. This procedure can be an external procedure or a set of operations executed on the database. In short, the action is the consequence of a trigger firing. In this architecture, the consequence of a trigger firing is of two types:

- **Action**: A transaction that is executed on the local database. The transaction usually includes operations on data that are part of the trigger state, or other variables from the environment. A typical trigger action is the update of a data variable. For instance, if trigger that has two position variables a and b as its inputs fires, its action can be the update of the data item that holds the distance between a and b. As mentioned before, since the execution model of this architecture is decoupled, the action transaction is separate from the transaction that generated the trigger event. It is inserted at the head of the internal transaction queue (see Section 3.6), and it is executed when its turn arrives. An action execution can generate data events that are themselves trigger events for other triggers.

- **Notification**: A trigger notification is the information that is sent to a subscriber to inform it that the trigger has fired. The trigger notification is an update of one or more data variables sent to a subscriber. A trigger notification can also be a message that simply informs a subscriber that the trigger has fired, without sending an update.

The trigger service maintains a persistent list of all triggers that are installed on the node. When a data event occurs as a result of the execution of a transaction, the event is signaled by the transaction service to the trigger service which, in turn, queries its list of triggers to check

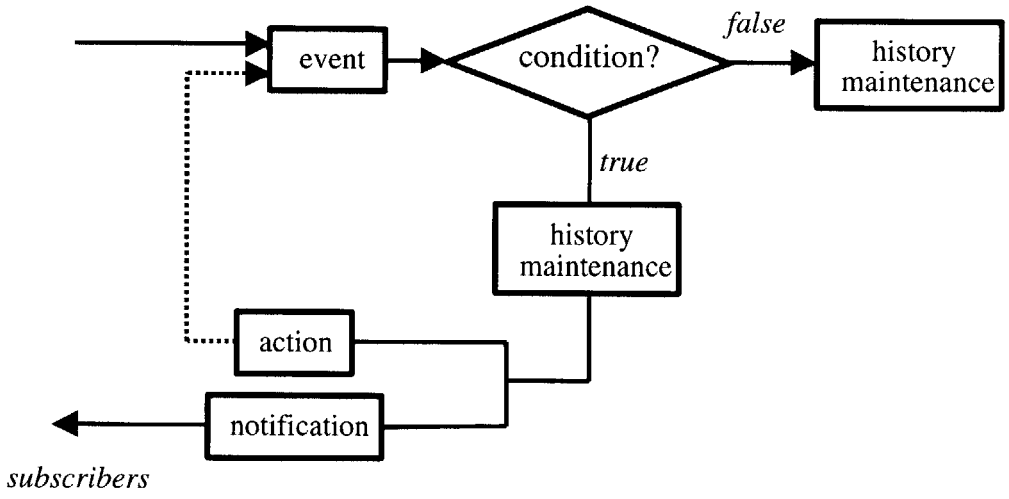


Figure 3-4: Trigger Execution Flowchart.

which triggers depend on this event (i.e. which triggers have the data event as a trigger event). For each of these triggers, the trigger service checks the condition. If the condition is evaluated to true, the trigger service updates the history, notifies the subscription service and adds the action transaction (if any) to the transaction service's internal transaction queue. If, on the other hand, the trigger condition is evaluated to false, only the trigger history is updated (see Figure 3-4).

3.8 Subscription Service

The *subscription service* provides two main types of functionality: It receives trigger subscriptions from clients, and notifies subscriber clients when triggers fire.

When a client is interested in being notified of a trigger firing, it subscribes to that trigger by sending a request to its own subscription service regardless of whether the trigger inputs are local, belong to the same remote node, or are distributed among different remote nodes.

When the trigger inputs all belong to the local node of the client, the subscription service simply adds the client's subscription to its persistent base of subscriptions. In this case, when the trigger service evaluates that trigger's condition to true upon the occurrence of a trigger event, the trigger service signals the trigger firing to the subscription service. The subscription

service notifies all the subscribers to that particular trigger.

The second case to consider is when the trigger inputs all belong to the same remote node (call it B). The subscription service of the host local to the client (host A) delegates the client's subscription to the subscription service of B. A trigger on the remote node B whose subscription is delegated for a local client on A is called a *remote trigger*. Notifications from the remote trigger are received as transactions through the transaction service of A. The subscription service of A, therefore, needs to react to this notification from B. This is achieved by installing on A a local trigger (called a *proxy trigger*) which has the notification from the remote trigger on B as its trigger event, and a constant condition that is always true. The proxy trigger will fire when a notification for the remote trigger arrives, in the form of an external transaction sent to A. Then, the subscription service of A notifies all the subscribers to the remote trigger. Note that at B, the subscription will appear as being initiated from the subscription service of A, and not a client on A. The subscription service of A, therefore, needs to subscribe only once to a certain trigger on B. This scheme clearly reduces the amount of communication needed for notifications since only one notification message is needed to be sent to a certain node no matter how many clients from that node are subscribers to a certain trigger. Further communication between the subscription manager and its local client subscribers is not a bottleneck since this communication does not (in general) use the limited-bandwidth network that connects nodes together. Note that updates that arrive as notification of remote triggers are effectively the replica of remote data. Trigger notifications, therefore, increase the availability of remote data to local clients (see Section 3.1.2).

The third case occurs when the trigger has inputs that belong to different nodes (remote or local). This case is a generalization of the second case described above. In this case, the trigger needs to be distributed among several nodes. One simple scheme for distributing the trigger requires the local subscription service to install the original trigger on the local node, and subscribe to a *default update trigger* τ_{x_i} on each node i that owns one of the trigger inputs x_i. τ_{x_i} has x_i as its input, and has a condition that is always true. In this case, upon the update of any trigger input x_i, the corresponding update trigger τ_i fires and notifies the local node of the update. The update of x_i will cause the local trigger's condition to be evaluated, and thus the local clients to be notified. In the case where the occurrence of a remote data

event is part of the trigger input, but an update of the input variable is not needed to evaluate the trigger condition, instead of installing a default update trigger, a *default event trigger* can be installed. A default event trigger is similar to a default update trigger, with the difference that the trigger sends a simple notification message instead of a variable update.

The following example illustrates the concept of remote triggers.

Example 3.1

Let x_A and x_B be the positions of nodes A and B respectively. Suppose a client on node A is interested in a notification whenever the distance between nodes A and B is greater than δx. A trigger $\tau_{|x_A - x_B| > \delta x}$ with such a condition reacts to updates on both x_A and x_B. Suppose the update of the local position x_A is obtained through a local client from a GPS system. To obtain updates of x_B, node A needs to install a remote trigger on B. The inputs for $\tau_{|x_A - x_B| > \delta x}$ are x_A and x_B , and the events that cause its evaluation are updates of either x_A or x_B . The condition is $|x_A - x_B| > \delta x$; the action is a simple notification.

The scenario for the subscription of such a trigger, $\tau_{|x_A - x_B| > \delta x}$, goes as follows. Assuming that the subscribing client knows the names of data elements, x_A and x_B, it sends a subscription request for $\tau_{|x_A - x_B| > \delta x}$ to the subscription service of its local node specifying the trigger inputs and condition. The subscription service checks if it already has a subscription for $\tau_{|x_A - x_B| > \delta x}$. If it does, it simply adds the client to the subscribers list. If not, then it installs $\tau_{|x_A - x_B| > \delta x}$ in the trigger base and adds the client as a subscriber; then it checks whether subscriptions for all remote inputs and condition arguments are available (x_B in this case). If not, then it subscribes to the default update trigger τ_{x_B} by sending a subscription request to node B's subscription service, and adds τ_{x_B} to the list of remote triggers that it has subscriber to.

3.9 Persistence Service

As was mentioned in Section 3.5, the database is viewed as another service in the architecture and not as the central component. The main purpose for using a persistence service is to allow a node that crashes to recover to a state as similar as possible to its state before crashing. This recovery capability is important for situations like the battlefield where system crashes

Service	Data Structure	Data Structure Name	Description
Transaction	queue	External Transaction Queue	Stores transactions submitted by external clients and remote nodes.
	queue	Internal Transaction Queue	Stores transactions generated by trigger actions.
	queue	Event Queue	Stores events that result from transaction execution.
Trigger	list	Trigger List	Stores triggers that are already installed on the node.
Subscription	hashtable	Subscribers HashTable	The key is a trigger, the value is a list of subscribers.
	list	Remote Trigger Subscriptions	Subscriptions to triggers on remote hosts already made.

Table 3.2: Persistent Data Structures Used by the Different Services.

may be frequent. It is also important in cases where the environment is very large so that it is impractical to store the state of the node in main memory.

In order to allow the system to recover, every service uses the persistence service to store either permanent or temporary information. Table 3.2 lists the different data structures that each service stores persistently. As one might conclude from this table and from the descriptions of the different services, the state of a node upon startup after a crash consists of all transactions, external and internal, that were submitted but not yet executed. It also contains all events that were generated by transaction execution but were not signaled to the trigger service, and all subscriptions that were submitted by local clients. Finally it contains the remote triggers that the local subscription service has subscribed to. This fine granularity in the persistent state is permitted by the highly uncoupled execution model of the architecture.

Figure 3-5 shows the different interactions between clients and services, and the interaction between the services themselves.

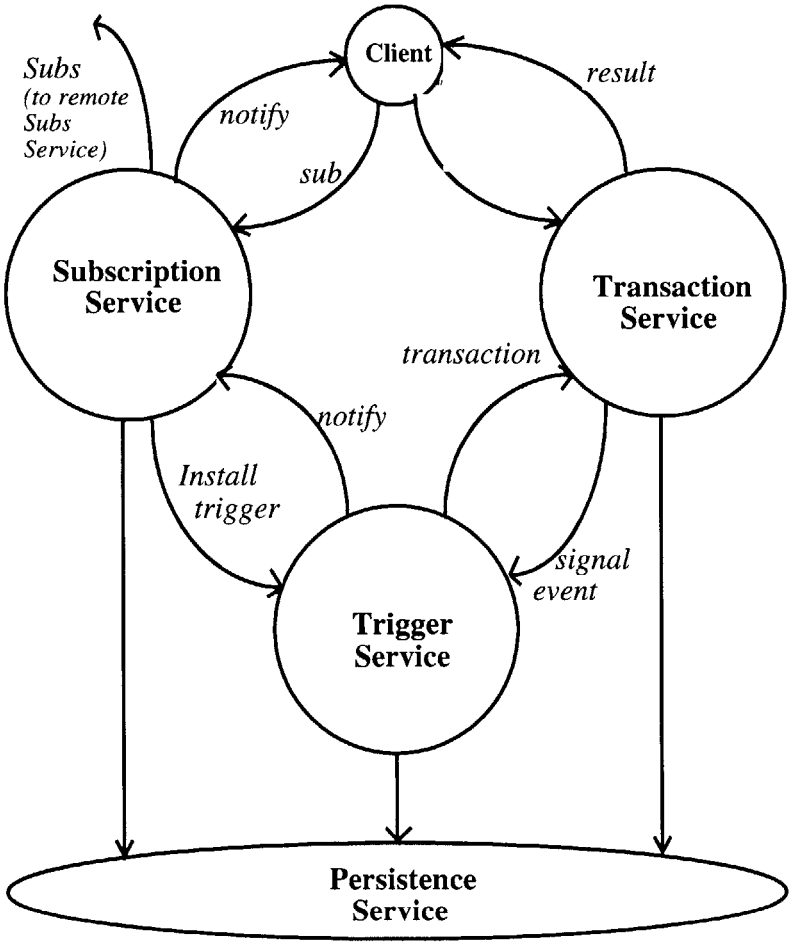


Figure 3-5: Interaction Between Different Services.

3.10 Higher Level Services

In Section 3.5, I mentioned several advantages of the service design used in this architecture. I conclude this chapter by describing another advantage of this design approach. As one might notice from Figure 3.5, the different services of the architecture are layered. The first layer consists of the Transaction Service and the Subscription service. The lower layer is the trigger service. The bottom layer is the persistence service, which, as I mentioned is used by the other services. Clients communicate with services in the first layer only. The trigger and persistence services are transparent to them.

The set of layered services described above can be viewed as an infrastructure for higher level services. More precisely, a set of triggers installed on one or more nodes can be seen as a service from the point of view of the subscriber. In Example 3.1, the two triggers $\tau_{|x_A - x_B| > \delta x}$ and τ_{x_B} together provide a service for the difference between the positions of x_A and x_B. Services involving more nodes with complicated triggers can be abstracted in this manner. In any case, as far as the subscriber is concerned, a subscription request to only one node is required. The local subscription manager has the responsibility of distributing remote triggers appropriately.

The subscriber, however, is still required to specify the details of the local trigger. These details can be hidden from the subscribers by implementing an additional service on top of the subscription service previously described. This service can publish a set of predefined abstract services (such as the position difference service of the example above). Published services can in this case be described to the subscribers qualitatively, instead of describing them in terms of triggers. Subscribers need not even know about the existence of triggers.

Subscribers might specify different levels of quality for their subscriptions, which translate to different argument values on the level of triggers (such as the value of δx in the position difference service). In our previous example, two clients might subscribe to the position difference service with arguments δx and $\delta x + \epsilon$, where $\epsilon \simeq 0$. In this case, two different triggers need to be installed, even though the two services are practically identical, especially with the presence of network delays that are comparable to the rates of change of positions. This problem can be avoided by allowing the arguments of published services to assume values that belong to a finite set that covers the range of meaningful qualities of service.

Higher level services can be seen as providing services with guarantees in the form of bounded

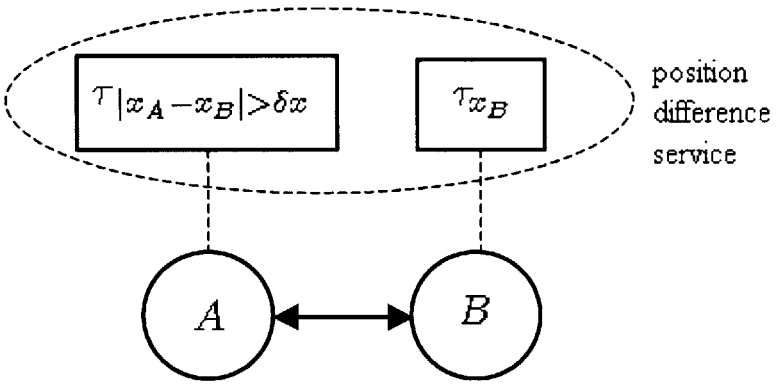


Figure 3-6: Triggers in Example 3.1 as a position differece service.

partial consistency. For instance, a trigger that sends an update of a position variable every 10 units guarantees to the subscribers that their version of the position variable is within 10 units of the true value. One should, obviously, take into account the network delays which increase inconsistency. Assuming a probabilistic model of delays on network links, it would be of interest to analyze what guarantees on consistency can triggers offer.

Chapter 4

Java Prototype: A Proof of Concept

4.1 Overview

This chapter describes the implementation of a prototype of the architecture presented in Chapter 3. The prototype is built in Java, and layers on top of existing Java services. The prototype does not implement all the architecture specifications presented in Chapter 3. In particular, it does not implement trigger histories. It also uses a simple scheme for distributing remote triggers. The prototype, however, does implement the Transaction, Trigger and Subscription services. It also implements the transactional semantics presented in Section 3.6.1. A standard method for building user defined triggers is also implemented.

The communication between clients and their local services, as well as the communication between different nodes uses Java Remote Method Invocation (RMI). RMI [Java98] is a standard Java service that provides a mechanism for invoking methods on remote objects and returning the invocation results, similar to CORBA. It hides the lower level complications of message passing using TCP sockets directly, thus allowing for simple object-oriented implementation of distributed applications.

The prototype uses ObjectStore PSE Pro for Java [ODI98] to implement the persistence service. ObjectStore is an object-oriented database suitable for light applications, that is, applications which need the DBMS for persistence, and do not need complex querying capabilities. ObjectStore provides transactional access to a local object-oriented database. It possesses the advantage of providing a Java Application Programming Interface (API) and thus directly

supporting the persistence of Java objects without translation.

The rest of this chapter describes the classes that implement names, transactions and triggers as well as the different architecture services.

4.2 Persistent Objects

A persistent object is an object that ObjectStore can store in the database. In ObjectStore terminology, such an object is called persistent-capable. In order for an object to be persistent-capable, it should implement the java.io.Serializable interface. Moreover, it should include fields that are either of basic Java types (i.e. int, float, String, etc.) or persistent-capable themselves. The ObjectStore API includes persistent-capable versions of Java Collection objects, which implement most common data structures.

As was mentioned in Section 3.9, different services use persistent data structures in order to allow system recovery. The system stores data objects along with these objects in the same database.

4.2.1 The Name Class

The architecture presented in Chapter 3 assumed that the data model is a hierarchy of objects, where each object has a unique owner node. Each object is, therefore, referenced by a unique name that reflect the position of that object in the object hierarchy. The naming scheme is implemented by the *Name* class. A Name object has two parts. The first part is the *owner*, which is a string that represents a host name of a valid IP address. The second part is a directory-like *path* that references the object within the host. Different levels of the path are separated by a '.'. A well formed path is of the form:

$$object\_name.member1.member2.member3... \tag{4.1}$$

The Name object has a string representation of the form:

$$//host\_name/object\_name.member1.member2.member3... \tag{4.2}$$

A persistent object in ObjectStore can be referenced using a unique string called a database root. A data object with a name such as that in (4.2) is stored in the database under the root name *hostname/object_name.*

This mapping between object Names and database root names implements the mapping between the abstract model and the database that was mentioned in Section 3.4.

4.3 Transaction Manager

The *Transaction Manager* class implements the Transaction Service described in Chapter 3. Before describing this class, I start this section by describing the transactional semantics that the prototype implements.

4.3.1 Transactions

Recall from Section 3.6.1 that a transaction is a batch of operations performed on data objects. The client builds a transaction by building an instance of the *MetaTransaction* object, which includes a description of the transaction. Objects that are involved in the transaction operation are registered in the transaction by adding them to a temporary hashtable with an arbitrary string as their key. Then operations are added to the MetaTransaction with a series of *addOperation* methods, each of which adds an operation of one of the types listed in Table 3.1. Registering an object in a transaction does not mean that the object becomes persistent. Object registration is a means for the transaction to reference the object in more than one operation, and a means for the client to access an object in the result of a transaction once this transaction is executed. The following example illustrates how to build a transaction that creates an object of type *Position* and reads an object of type *map*:

//create data object

Position myPosition = new Position(5,4);

//create the MetaTransaction object

MetaTransaction myMetaTransaction = new MetaTransaction();

//registering the position object under the hash key "item1"

myMetaTransaction.register("item1", myPosition);

//create a name referring to the position object in the database

Name name1 = new Name("ceci.mit.edu/myPosition");

//adding an operation to create the database object myPosition

myMetaTransaction.addCreateOperation(name1, "item1");

//create a name referring to the map object in the database

Name name2 = new Name("ceci.mit.edu/myMap");

//add an operation to read the database object myMap and register it under the hash key "item2"

myMetaTransaction.addReadOperation(name2, "item2");

In order to better explain how a client submits a transaction, I first briefly describe the *client* class.

An application that is to operate as a client in the architecture should extend the client class. The client class is constructed with a client name and with the name of its local Transaction Manager. Since the client and the Transaction Manager communicate using RMI, these names are used by RMI to identify both objects.

Two types of transactions can be submitted by the client. A blocking transaction blocks the client thread until the transaction result is returned from the Transaction Manager. A non-blocking transaction only blocks the client until the transaction itself is made persistent by the Transaction Manager, by adding it to the external transaction queue; that is the transaction has been accepted for later execution. In this case, a client can still refer to the result of its transaction, by checking if the Transaction Manager has returned it (asynchronously). A client submits the transaction of the above example as blocking in the following manner:

Client myClient = new Client(clientName, TransactionManagerName);

myClient.initialize( );

TransactionResult myBlockingResult =

myClient.submitTransaction(myMetaTransaction);

The map object that was read in the transaction can be accessed from the *TransactionResult* object. A non-blocking transaction can be submitted as follows:

myClient.submitTransaction(myMetaTransaction, "myNonBlockingResult");

Later on, the client can check if the result is obtained by performing the following:

TransactionResult myNonBlockingResult =

myClient.getResult("myNonBlockingResult");

Setting the second argument of the submitTransaction( ) method to null indicates to the Transaction Manager that the client is not interested in the transaction result. In the case where a client transaction is aborted, the client can obtain this information from the *transactionResult*, together with a description of the reason why the transaction failed.

Note that both *submitTransaction* methods call RMI methods in the TransactionManager class.

The TransactionManager class implements the Transaction Service described in Section 3.6. The members and methods of this class are static, which ensures that a Java Virtual Machine (VM) can only run one instance of this class. Given this fact, the instance of the TransactionManager running on a Java VM is referred to as "the Transaction Manager". The Transaction Manager is responsible for accepting transactions from clients, executing them, returning their results and signalling events that could activate triggers to the Trigger Service implementation.

A MetaTransaction contains the description of the client transaction. It is the task of the Transaction Manager to translate this description to actual operations on the database, and execute them atomically using ObjectStore's transactional capabilities. ObjectStore, however, allows only one transaction to be active at any time. The approach taken to solve the problem of both accepting client transactions and executing them is to open a global transaction when the Transaction Manager is initialized, and then regularly apply a *checkpoint* operation to this transaction when necessary. The checkpoint operation amounts to committing the transaction and leaving it open. However, the checkpoint operation commits all persistent-capable objects present in the VM to the database as persistent objects. This includes both data objects and manager objects (such as transaction queues). The synchronization of transaction execution is done as follows: as long as there are transactions present in either the external transaction queue or internal transaction queue of the Transaction Manager, a thread in the Transaction Manager pops the queue, executes the transaction and applies a checkpoint operation on the global ObjectStore transaction. Meanwhile, a client that submits a transaction should not return until its transaction is made persistent in the external transaction queue. It should

also not commit the global transaction itself, since a commit while the Transaction Manager is executing another client transaction is unsafe. The client, therefore, submits its transaction in a thread that adds itself to a thread group in the Transaction Manager and waits. When the transaction execution thread executes a transaction, it checks all waiting "submit threads", makes their transactions persistent and returns them. Then the execution thread executes the next transaction, if any. Submit threads that contain non-blocking transactions are returned immediately after their transaction is made persistent in the external transaction queue. Those that contain blocking transactions are only returned when the transaction is executed and its result is obtained. The following section presents the synchronization algorithm is in a pseudo-code form.

Transaction Manager Algorithm

As was mentioned above, the client submits its transaction in a different Java thread called the submit_thread. The submit_thread object contains a number of flags that indicate its state. The isBlocking flag indicates whether the transaction is blocking or not. The added flag indicates whether the transaction contained in the thread was added to the external transaction queue or not. The resultObtained flag indicates whether this thread's transaction is already executed (in the case where the transaction is blocking). In the SUBMIT_TRANSACTION method, the submit_thread object is added to a list of threads and its flags are initialized. Then it waits until its returned flag is set to true.

```
SUBMIT_TRANSACTION( newTransaction, isBlocking ) {
  submit_thread ← new Thread (thread_list[ ], newTransaction)
  submit_thread.returned ← FALSE
  submit_thread.added ← FALSE
  submit_thread.blocking ← isBlocking
  sumbit_thread.resultObtained ← FALSE
  do submit_thread.WAIT( ) until submit_thread.returned = TRUE
   if submit_thread.blocking = TRUE
     return submit_thread.transaction.result
}
```

The Transaction Manager runs a separate thread that adds client transactions to the external transaction queue, and executes transactions as they are present in queue:

```
PROCESS_THREAD {
   while( TRUE )
     if ( internal_queue.length > 0 )
       PROCESS( internal_queue )
     else if ( external_queue.length > 0 )
       PROCESS( external_queue )
     global_transaction.CHECKPOINT( )
     RETURN_THREADS( thread_list[ ] )
}
PROCESS( queue ) {
     current_transaction = DEQUEUE( queue )
     current_transaction.EXECUTE( )
     if ( current_transaction.aborted = TRUE )
          global_transaction.BEGIN( )
          DEQUEUE(queue)
          global_transaction.CHECKPOINT( )
     else
          internalQueue.ADD( SIGNAL_EVENTS( ) )
}
RETURN_THREADS ( thread_list[ ] ) {
 for i ← 0 to thread_list.length
     if thread_list[i] != NULL
          if thread_list[i].added = FALSE
                    thread_list[i].added ← TRUE
                    externalQueue.ADD( thread_list[i].transaction )
                    if thread_list[i].blocking = FALSE
                         thread_list[i].returned ← TRUE
           else if thread_list[i].added = TRUE and  thread_list[i].blocking = TRUE
```

```
                    if thread_list[i].resultObtained = TRUE
                        thread_list[i].returned ← TRUE
    global_transaction.CHECKPOINT( )
    }
```

The PROCESS_THREAD thread loops forever, checking if the internal transaction manager contains transactions that have not been executed yet. If not, it checks if the external transaction queue contains a transaction. This gives priority to internal transactions. When a transaction is found in either of the two queues, the transaction is executed in the PROCESS method. The events that are generated from the execution of the transaction are passed to the Trigger Manager using the *signalEvents( )* method. This method returns any transactions that are generated by triggers firing. These transactions are added to the internal transaction queue.

At every round of the infinite loop, the list of submit_thread objects is checked to see if there is a submit_thread that can be returned. This is done in the RETURN_THREADS method. A submit_thread can be returned in one of two cases:

- If it is non-blocking and it was added to the external transaction queue (indicated the added flag).
- If it is blocking, it was added to the transaction manager queue, and its result was obtained (indicated the resultObtained flag).

If the RETURN_THREADS method finds a transaction that was not added it adds it to the external transaction queue, and only returns it if it is non-blocking. If, on the other hand, the transaction was added, it is blocking and its result was obtained, it is returned by setting the thread's returned flag to true.

4.4 Trigger Manager

This section describes the *TriggerManager* class that implements the Trigger Service described in Section 3.7. However, before discussing this implementation, a description of trigger implementation is presented.

	Member	Description
	events	Vector of Names of variables which are events that cause the trigger to be evaluated.
	hasEvent(Name event)	Returns true if the trigger has the Name in the argument as an event.
	getEvents( )	Returns the trigger events.
	conditionTransaction	A MetaTransaction object containing transaction required to evaluate the trigger condition.
	getConditionTransaction( )	Returns condition transaction.
abstract	*evaluateCondition(TransactionResult tr)*	Evaluate the trigger condition. The information needed for evaluation is contained in the tr argument. This result is obtained by executing the transaction contained in *conditionTransaction.*
	actionTransaction	A MetaTransaction object that holds the transaction that is executed on the local database when the trigger fires.
	getAction( )	Returns the *actionTransaction.*
	notification	Name of data object that is sent to subscribers when the trigger fires.
abstract	*toString()*	Returns a unique string representation of the trigger.
abstract	*equals(Trigger T)*	Returns true if the string representation of the trigger is equal to that of the argument.
	copy()	Returns a copy of this trigger instance.

Table 4.1: Trigger Class Description.

4.4.1 Triggers

The prototype allows users to install fairly arbitrary triggers. However, in order for these triggers to operate correctly and in order for the architecture to interface correctly to triggers, new trigger classes are required to have a standard structure. This structure is defined by an abstract Java *Trigger* class, which defines a skeleton of members and methods that user-defined triggers should implement. New triggers are thus classes that extend the Trigger class. Table 4.1 lists the members and methods that the Trigger class contains.

An event that can potentially fire a trigger is bound to a specific object. Trigger events are, therefore, specified in a vector of Name objects that hold the names of these data objects. An event is thus bound to an object through that object's. The trigger condition is implemented in two parts. The *conditionTransaction* member is a MetaTransaction that is executed as a precondition fot the evaluation of the trigger condition. The *TriggerManager* executes this

transaction to obtain values of data items that it needs to evaluate the condition. This transaction, therefore, should include read and invoke operations only. The result of this transaction is passed to the *evaluateCondition( )* method, which evaluates the condition of the trigger and returns the result as a boolean value. The trigger action is a transaction (of type MetaTransaction) that is executed on the local database when the trigger condition is evaluated to true. The *notification* field contains the Name of the data object that is sent to subscribers when the trigger fires.

The following example illustrates a trigger that takes a position object as its input, and fires when the distance between the new value of the position and the last updated value exceeds a certain threshold *delta*. The trigger class name is *PosDiffFromLastTrigger*. In addition to the name of the input position, the trigger requires two other arguments: a threshold *delta* and a position object to hold the value of the last position update:

```
protected Name posA;
protected Name posAlast;
protected double delta;
```

The constructor adds the name of the input position to the event vector. It also assigns the name of the notification object, and constructs the condition and action transactions:

```
public PosDiffFromLastTrigger(Name posA, Name posAlast, double delta)
extends Trigger {
    events.add( posA );
    this.posA = posA;
    this.posAlast = posAlast;
    this.delta = delta;
    //notification is the current position
    notification = posA;
    //construct conditionTransaction,
    //the conditions needs the values both the current and last positions.
    conditionTransaction = new MetaTransaction( );
    conditionTransaction.addReadOperation( posA, "posA" );
    conditionTransaction.addReadOperation( Alast, "posAlast" );
```

```
    //construct actionTransaction,
    //this transaction reads the value of the new positions and assigns its
    //value to the last position variable.
    actionTransaction = new MetaTransaction( );
    actionTransaction.register( "posA", posA );
    actionTransaction.addReadOperation( posA, "posA" );
    actionTransaction.addUpdateOperation( posAlast, "posA" );
}
```

The *evaluateCondition( )* method obtains the values of both current and last positions, and checks if their difference is larger that *delta:*

```
public boolean evaluateCondition(TransactionResult tr) {
    Position pA = (Position)(tr.getResult( ).get( "posA" ));
    Position pAlast = (Position)(tr.getResult( ).get( posAlast ));
    return ( pA.distanceFrom( pAlast ) > delta ) ? true : false;
}
```

Finally, every trigger class should implement the *toString( )* method. This method returns a unique string which represents the trigger object. The convention is for the string representation to start with the name of the trigger subclass, followed by arguments that distinguish the trigger object. For instance, if the trigger in the previous example is instantiated as follows:

```
posA = new Name("tank1.pos", "ceci.mit.edu");
posAlast = new Name("tank1.posLast","ceci.mit.edu");
PosDiffFromLastTrigger t1 = new PosDiffFromLastTrigger(posA, posAlast, 0.2);
```

Then the string representation of trigger t1, as returned by *toString( )* is:

```
PosDiffFromLastTrigger://CECI.MIT.EDU/tank1.pos:
    //CECI.MIT.EDU/posAlast:0.2
```

It is clear from this example that two triggers have the same string representation if and only if they have the same position arguments and the same delta threshold, and, therefore, are equivalent. Consequently, two triggers are equal (i.e. the *equal( )* method returns true) if they have the same string representation.

A distinction should be made between a trigger class and a trigger instance. A trigger class

is a Java class that extends the Trigger class, such as the PosDiffFromLastTrigger class in the example above. On the other hand, a trigger instance is an instantiated object of a trigger class, such as the object t1 in the example above. Clearly, two trigger instances of the same trigger class can have different arguments. In this case, they have different string representations.

As in the case of the TransactionManager class, the fields and methods of the Trigger-Manager class are static. This class is, therefore, referred to this class simply as "the Trigger Manager".

As far as a client is concerned, installing a new trigger instance on the system and subscribing to one that already exists are the same. The client submits a subscription that includes the trigger instance and the client's Name to the Transaction Manager. The subscription is submitted in a *submitThread* similar to that of a transaction. This submission is synchronized through the same mechanism described in Section 4.1, that is, it waits until it is added to the external transaction queue. When its turn for processing arrives, the Transaction Manager detects that it is a subscription request, and therefore submits it to the Trigger Manager. The Trigger Manager, which maintains a persistent list of all triggers existing at the node, checks whether an identical instance of the trigger exists in this list (using the trigger's *equal( )* method). If it does not, it adds the instance in the subscription request. In both cases it submits the subscription request to the Subscription Manager (see next section).

When the execution of a transaction results in the generation of events, the Transaction Manager calls the Trigger Manager's *signalEvents( )* method repeatedly with generated events (see page 50) . The Trigger Manager checks which triggers in its list have the specified event. For each trigger that qualifies, the Trigger Manager obtains the *conditionTransaction* and performs it. Note that the *conditionTransaction* does not have to go though the transaction queue of the Transaction Manager; the Trigger Manager can perform it directly (using the Transaction Manager's API), since the Transaction Manager is not executing other transactions at the time when the *conditionTransaction* is executed. The Trigger Manager obtains the result of the *conditionTransaction*, and calls the *evaluateCondition( )* method with the result as an argument. The *evaluateCondition( )* method uses this result to evaluate the actual condition function, and returns a boolean.

In the case where the trigger's *evaluateCondition( )* returns true, the Trigger Manager notifies the Subscription Manager. The trigger's *actionTransaction*, if any, is stored in a temporary queue. When all the triggers that have the current event have been checked, this temporary queue is returned by the Trigger Manager's *signalEvents( )* method. The Transaction Manager appends this queue to the head of its own internal transaction queue, and resumes execution. Since the internal transactions have higher priority (see Section 3.6), the Transaction Manager will process the transactions resulting from a trigger that fires before executing other transactions that exist in its external transaction queue.

4.5 Subscription Manager

The *SubscriptionManager* class, which implements the Subscription Service is also a static class, and therefore I refer to it as "the Subscription Manager". The Subscription Manager is responsible for two main tasks: accepting subscriptions and sending notifications to subscribers.

As mentioned in Section 3.8, a client only subscribes to its local Subscription Manager. The Subscription Manager has the responsibility of delegating remote trigger subscriptions to the appropriate remote Subscription Managers. The Subscription Manager maintains a persistent hashtable of subscriptions called the *subscriptionsHashtable.* The keys in this hashtable are string representations of triggers, and the list of subscribers is the corresponding value. When a client submits a subscription request to the Subscription Manager, the Subscription Manager installs the subscription in one of the three ways mentioned in Section 3.8. In the case of a trigger having inputs that belong to different remote triggers, the Subscription Manager implements the simple strategy of installing default update triggers on remote hosts. It then adds the original trigger to the *subscriptionsHashtable* with the client's Name.

Obviously, in all three cases, when the trigger instance exists in the *subscriptionsHashtable*, the client's Name is added to the list of subscribers to that trigger. This explains the fact that to a client, trigger installation and subscription appear to be the same. The only requirement is that for every trigger that is to be installed on the system, the trigger class should be in the class path of the Java VM that runs the three services. It should also be mentioned that the Subscription Manager maintains a list of remote triggers that it has subscribed to. This

prevents it from subscribing twice to the same remote trigger, and thus saves unnecessary communication.

A notification to subscribers occurs when a trigger fires. In this case, the Trigger Manager signals to the Subscription Manager that the trigger fired. The Subscription Manager obtains the object indicated by the *notification* Name of the trigger, and sends the notification to subscribers using RMI. In the case where the subscriber is a client, the Subscription Manager calls a client RMI method called *getNotified( )*. The Subscription Manager passes to this method the value of the notification object, and the string representation of the trigger that caused the notification. This string representation is used by the client to identify the notifying trigger. On the other hand, when the subscriber is a Subscription Manager of a remote trigger, the notification is sent as an update operation in a MetaTransaction to the Transaction Manager of the same remote host. The execution of this transaction results in the firing of the proxy triggers that the remote node has installed on itself. This, in turn, results in clients on the remote trigger being notified of the update.

A notification transaction is sent in a separate thread from the main thread that executes transactions and manages triggers (this is done whether the subscriber is a client or a remote Subscription Manager). This serves two purposes. First, it prevents potential deadlocks. For instance, assume that node A is notifying node B, and at the same time B is notifying A. Then A will be waiting for its call to submit its notification transaction to return, but B cannot return it because it is waiting for its own submit call to return from A. Submitting notifications in different threads prevents such a deadlock. Second, when the subscriber is a client, notifying him in a separate thread reduces the time where the main thread of the system is blocked and also prevents malicious behavior from the client's side, such as intentionally blocking the Subscription Manager.

Chapter 5

Conclusion

5.1 Summary of Work

This thesis presents a trigger architecture to implement information distribution in distributed low bandwidth environments where information change is dynamic. The main purpose of the architecture is to automate information distribution in the battlefield at the lower echelon level. The design principles of the architecture, however, are appropriate to implement automated information distribution in other similar environments such as the stock market and civilian air traffic control.

The environment is assumed to consist of a set of hosts connected by a communication network. Each host possesses data items that only it can update. It is assumed, however, that clients present on a certain host may be interested in data that belongs to hosts other than their own. Since complete consistency is neither possible nor desirable in situations where data change dynamically and available network bandwidth is restricted, only relevant changes to data are sent to interested clients. Clients express their interest in data items by subscribing to triggers. Trigger conditions implement a mechanism to decide when a relevant change in data occurs and subscribers should be updated. Every host, therefore, contains a partially consistent model of the environment, and triggers are a mechanism to manage the inconsistency of the model under certain network conditions.

The thesis argues that the traditional approach of active database systems does not present a ready solution to this problem. It advocates for building the architecture as a set of services.

A host's model is stored in a local database managed by a persistence service. Local clients access their local model by submitting transactions. A transaction in this case is a batch of operations that is executed atomically. The transaction service receives client transactions and executes them in serial order. Triggers are managed by the trigger service. The trigger service is responsible for executing triggers when trigger events occur. Client subscriptions to triggers are managed by the subscription service. The subscription service delegates trigger subscriptions initiated by remote clients to the subscription services of the respective remote hosts. It also notifies subscribers when triggers fire. The persistence service is used by the other three services to store their state (i.e. transactions, triggers, subscriptions) persistently. This allows a host to recover from crashes and restore itself to a state similar to that it was in before crashing.

Chapter 4 describes a Java implementation of the architecture using ObjectStore PSE Pro for persistence. Transactional semantics that build on top of ObjectStore's transactional capabilities are developed. In addition the prototype allows users to define arbitrary triggers in a manner that conforms to a standard structure defined by an abstract Trigger class. Clients subscribe to triggers and receive trigger notifications through an implementation of the subscription service. Communication between clients and services, as well as communication between services on different hosts is carried through Java RMI.

5.2 Future Work

5.2.1 Naming and Directory Services

The Java prototype in Chapter 4 uses a static naming scheme built on top of Java RMI. A general implementation of the architecture requires a consistent naming scheme to translate between the real environment and data objects that serve as inputs to the triggers. The situation is further complicated by the fact that the name mapping is not stable, since host addresses and network routing can change dynamically. Furthermore, the hierarchy of data items (see Section 3.4) may also change dynamically during the operation of the system. Directory services complement naming services in implementing this dynamic hierarchy.

The Java Naming and Directory Interface (JNDI) [Lee99], a Java Standard Extension, is a platform and protocol independent layer that unifies the interface to a number of currently

supported naming and directory services. JNDI is relatively semantic free and independent of any specific directory service implementation, but it does support the concept of federated name spaces. JNDI is an appropriate implementation for a Naming service that could be integrated with the other architecture services.

5.2.2 I/O Automata Formulation of Triggers

The *input/output (I/O) automaton* model is a general tool for modeling asynchronous distributed and concurrent systems. It was first introduced in [LT87]. The model has very little structure by itself, which provides for its flexibility. However, it provides a framework for reasoning about correctness and performance of distributed systems. [Lyn96] includes a detailed description of the model.

Triggers, as presented in Section 3.7.1, can be modeled as I/O automata with trigger events as the automaton input action, trigger's history as the automaton's state, condition evaluation as the automation's internal action and trigger notification as the output action. Such a formulation allows a more precise formulation of the higher level services presented in Section 3.10 through the composition of trigger automata. Moreover, a modeling of the network links as channels with probabilistic delays allows one to quantify the partial consistency that triggers can guarantee.

A formal model of the transaction service facilitates the development of a transaction execution model which is more efficient than that presented in Section 3.6.

5.2.3 Trigger Distribution

Section 3.8 argued that remote triggers should be installed on remote hosts to implement a trigger that depends on data objects that belong to different nodes. It also introduced a simple scheme for remote trigger installation. This scheme, in effect, causes every update of data items that constitute the trigger's input to be pushed to the trigger host. Such a scheme is not efficient since partial trigger conditions can be evaluated on hosts that are closer to the input sources than the trigger host is. The trigger node is thus only updated when the update affects the trigger's condition. Trigger distribution schemes that minimize different objective functions might be developed. One possible objective function is the expected number of messages sent

when a trigger event happens. The frequency of occurrence for each trigger event clearly affects the optimal scheme for distributing a certain trigger among the network hosts.

A final aspect of the problem under current consideration analyzes the effect of optimal trigger distribution on the amount of partial consistency that triggers can guarantee.

Bibliography

[ACM96] The ACT-NET Consortium. "The Active Database Management System Manifesto: A Rulebase of ADBMS Features", *ACM Sigmod Record, 25:3,* September 1996.

[AG89] R. Argawal, N. H. Gehani. "ODE (Object Database and Environement): The Language and the Data Model". In *Proc 1989 ACM-SIGMOD Conf. on Management of Data*, Portland, Oregon, June 1989.

[AMO93] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, *Network Flows.* Prentice-Hall, NJ 1993.

[BB+98] A. Bailey, I. Bazzi, V. J. Harward, J. C. Lopez, S. Mneimneh and R. Zbib, "An Improved Hierarchical Caching Architecture for Low Bandwidth Networks," In *Proc. 2^{nd} Annual FedLab Symposium;* 11-16, College Park, MD. February 1998.

[Cha95] S. Chamberlain, "Model-Based Battle Command: A Paradigm Whose Time Has Come", *1995 Symposium on C2 Research and Technology,* NDU, June 1995.

[COD73] CODASYL Data Description Language Committee. *CODASYL Data Description Language Journal of Development,* NBS 113, June 1973.

[CS96] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP, vol III, Client-Server Programming and Applications.* Prentice-Hall, Inc. New Jersey, 1996.

[Day95] U. Dayal, "Ten years of activity in active DBS: What have we accomplished?", *Proc. Active and Real-Time Database Systems.* Skövde, Sweden. 1995.

[Gat96] S. Gatziu, "The SAMOS Active DBMS Prototype", *Proceedings of the 6th Hellenic Conference in Informatics,* 1997.

[GK+98] S. Gatziu, A. Koschel, et al. "Unbinding Active Functionality", *ACM Sigmod Record* 27:1, March 1998.

[IBM78] IBM. IMS/VS *Application Programming Reference Manual.* SH20-9026, IBM, White Planes, NY, 1978.

[Java98] Javasoft, JAVA^{TM} *REMOTE METHOD INVOCATION (RMI) Interface*, available at http://www.javasoft.com/products/jdk/rmi/index.html.

[KG+98] A. Koschel, S. Gatziu, H. Fritschi, G. von Bültzingslöwen, "Applying the Unbundling Process on Active Database Systems", *Intl. Workshop on Issues and Applications of Database Technology (IADT)*, Berlin, Germany, July 1998.

[Lee99] R. Lee, *The JNDI Tutorial: Building Directory-Enabled Java Applications*, available at http://java.sun.com/products/jndi/tutorial.

[LM+94] N. A. Lynch, M. Merritt, W. Weihl and A. Fekete, *Atomic Transactions.* Morgan Kaufmann Publishers, San Mateo, CA 1994.

[LT87] N. A. Lynch and M. R. Tuttle, *Hierarchical correctness proofs for distributed algorithms.* Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, April 1987. Technical Report MIT/LCS/TR-387.

[LT88] Nancy A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *Technical Memo MIT/LCS/TM-373*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, November 1988.

[Lyn96] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Fransisco, CA 1996.

[MD89] D. McCarthy and U. Dayal. "The architecture of an active data base management system," in *Proc. ACM-SIGMOD Conf. on Management of Data*, Portland, Oregon, June 1989.

[Mor83] M. Morgenstern, "Active Databases as a Paradigm for Enhanced Computing Environments," in *Proc. 9th Inernational Conference on Very Large Database Systems,* 34-42. Florence, Italy. 1983.

[ODI98] Object Design, *Bookshelf for ObjectStore PSE/PSE Pro Release 3.0 for Java*, available at http://mothra.odi.com/content/products/pse/javadoc/index.html.

[OMG91] Object Management Group, *The Common Object Request Broker: Architecture and Specification,* OMG Document Number 91.12.1, Revision 1.1, December 1991, available at http://www.omg.org/pub/docs/1991/91-12-01.pdf.

[OMG97] Object Management Group, *CORBA services specification,* formal/98-07-05, Chapter 4, available from http://www.omg.org/pub/docs/formal/97-12-11.pdf.

[PD+93] N. W. Paton, O. Díaz, M. H. Williams, J. Campin, A. Dinn and A. Jaime, "Dimensions in Active behavior," in *Proc. 1st International Workshop on Rules in Database Systems*, Edinburgh, Scotland, August 1993.

[PT+97] G. Pappas, C. Tomlin, J. Lygeros, D. Godbole and S. Sastry, "A Next Generation Architecture for Air Traffic Management Systems," in *Proceedings of the 35th Conference on Decision and Control*, San Diego, CA, 1997.

[WC96] J. Widom and S. Ceri, eds. *Active Database Systems.* Morgan Kaufmann Publishers, San Fransisco, 1996.

[Zlo77] M. M. Zloof, "Query-by-Example: A Data Base Language", *IBM Systems Journal*, 16(4):324-343, 1977

[ZM+99] R. Zbib, S. Mneimneh, J. C. Lopez, V. J. Harward and R. Rabbat, "The Trierarch Trigger Architecture," in *Proc. 3rd Annual FedLab Symposium,* 267-271, College Park, MD. February 1999.