A GENERALIZED FRAMEWORK FOR MONITORING CHANGES TO DATABASE RELATIONS USING ACTIVE CAPABILITY

The members of the Committee approve the master's thesis of Levette Stephen Lobo

Sharma Chakravarthy
Supervising Professor _____

Leonidas Fegaras _____

David Kung _____

To my family.

A GENERALIZED FRAMEWORK FOR MONITORING CHANGES TO DATABASE RELATIONS USING ACTIVE CAPABILITY

by

Levette Stephen Lobo

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2004

ACKNOWLEDGEMENTS

I express my sincere gratitude to Dr. Sharma Chakravarthy, for the opportunity to work on this topic. His understanding, insight and continual guidance made this work a rewarding learning experience.

I would like to thank Dr. Leonidas Fegaras and Dr. David Kung for their time and interest in serving on my committee.

My gratitude goes to Raman Adaikkalavan, Ganesh Gopalakrishnan, and Ambika Srinivasan for their help and advice during the implementation of this work. I am thankful for the camaraderie and encouragement of Alpa Sachde, Ajay Eppili, Laali Elkhalifa, Manu Aery, Shravan Chamakura and all my wonderful friends at the ITLAB.

My deepest appreciation and love goes to my parents and sister for their encouragement, faith, unconditional love and support.

This work was supported by the Office of Naval Research, the SPAWAR System Center-San Diego, the Rome Laboratory (grant F30602-01-0543), and the NSF (grants IIS-012370 and IIS-0097517).

April 7th, 2004

ABSTRACT

A GENERALIZED FRAMEWORK FOR MONITORING CHANGES
TO DATABASE RELATIONS USING
ACTIVE CAPABILITY

Publication No. _____

Levette Stephen Lobo, M.S.

The University of Texas at Arlington, 2004

Supervising Professor: Sharma Chakravarthy

Current data-driven applications operate in a global context with shared data being modified by several applications. Database management systems (DBMSs) provide mechanisms for sharing large amounts of data and manipulating them by many concurrent users in a consistent manner. However, they were not designed to support monitoring of subsets of relations by multiple users when they are being changed or manipulated by several users. In many applications, it is not only necessary to monitor these subsets but also to visualize the changes in some way. For example, in one of the applications, the track relation is being updated with respect to tracks (or objects) and their positions by multiple sources. Many users monitor subsets of these tracks as needed for their use (e,g, different regions).

The objective of this thesis is to overcome the limitations of current DBMSs with respect to change monitoring and notification. We present an approach that leverages active technology to enable dynamic change monitoring of database relations in an efficient

and transparent manner. In particular, we extend a mediator based Active Technology system, the ECA Agent, and discuss its architecture, design and implementation details.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Motivation

There exists a large class of computing and information processing applications that operate on shared data and operate in a global context: they exchange information from distributed locations between each other. Although some of these client applications access static data, most others add to, modify or remove from some subset of dynamic, shared data. One requirement in this multi-client environment, where large data repositories are involved, is that certain clients would like to receive notifications of changes made to a subset of this data. In essence they wish to answer the question: ***"Who did what to which data and when?"***. A number of concrete applications that exhibit this requirement are discussed below.

1.1.1 Monitoring applications

Consider a package delivery service such as FedEx or UPS. The location and status of packages is constantly updated in the distribution network's computing facilities. Whenever a package is in a different stage of the delivery, such as, *in-transit* or *delivered*, an update to its status is made in the network's servers. Recipients of the package or any interested party would like to be intimated of any change in status on a continuous basis, particularly if the package is of critical importance. There will likely be several clients interested in a set of packages. In this scenario we have multiple clients that wish to monitor only some dynamic information which is being updated by clients of the same system.

Tracking applications are an important class of applications where monitoring capability would be particularly useful. In one specific case, SPAWAR, San Diego, has developed a track application that uses the ECA Agent to actively observe the status of various objects in a particular air space. These include friendly and enemy aircraft. The locations of these object are continuously updated by several tracking stations. There was a need to be able to centrally monitor the updates of all of these stations to particular subsets of data within their databases. Child locator systems, such as Wherify's GPS Personal Locator [1] is another viable application. Updates of the geographical position of the locator are sent at regular intervals to a central server. Wherify operators alert the child's parents, who can log onto a web site that requires user identification and pinpoint their child's location. A similar application may be used to monitor the location of vehicles transporting hazardous cargo or sensitive material. Tracking devices periodically send updates of their current location and status to a central authority. Once again we see that this centralized system is propagating the updates of information from one set of clients to another set of observing clients.

Another class of applications is more concerned with the ***security and profiling*** of the data itself. For example, in addition to the measures in place to ensure authorized access and data integrity, a database administrator (DBA) may wish to monitor critical relations, without disrupting the activities of modifying clients. The transparent nature of this monitoring activity is a critical requirement. As another instance of this class, many databases make updates to system catalogs, which are also stored as relations in the database. It is possible, and even likely, that a DBA may wish to closely observe these system statistics on a continuous basis through an external interface to determine whether the database is operating within acceptable limits. In essence, the questions that all these classes of monitoring applications seek to answer are:

- Who is the modifier, as identified by the database system?

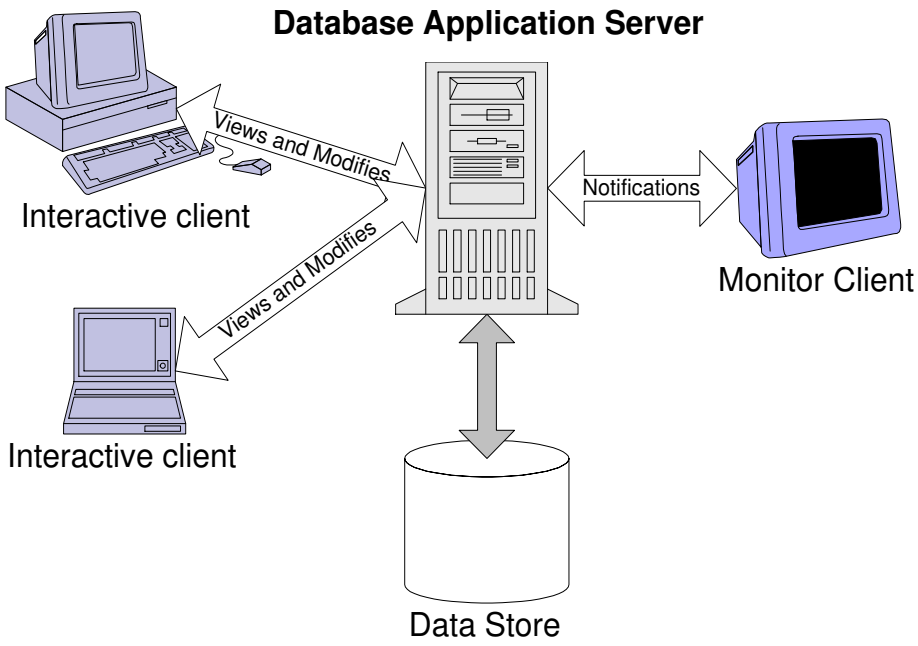

Figure 1.1 Application Scenario with Interactive and Monitoring Clients.

- What modifying actions were caused by that user against data records of interest?
- What are the records in the subset of interest that were affected by the modifier's actions?
- How recent are these changes?

1.1.2 Application characteristics

There are several traits for the classes of applications mentioned here. First, they involve two sets of client applications: one set is composed of distributed, **"interactive"** clients updating a centralized data repository in the course of their execution; the other set is comprised of distributed clients simply observing changes to some information specific to their interest or requirements. The latter set, or **"monitoring"** clients, require notifications whenever the data, that they are interested in, changes. This is illustrated in figure 1.1.

Second, given the nature of the applications, there will be large data sets involved. The task of managing these large volumes of data is typically offloaded to a relational database management system (RDMBS) in order to reap their associated benefits. Thus, all modifications made by applications, having some interest in the shared data at different logical levels, eventually filter down to common data modification language (DML) operators such as SQL INSERT, DELETE and UPDATE commands acting upon some subset of the underlying relations in the logical database schema. Third, the notifications must be propagated to the subscribing client as quickly as possible.

Finally, the monitoring clients have dynamic monitoring interests. In particular, they may be interested in the changes made by *a set of clients* to some subset of data that they are authorized to monitor. Both the specified set of clients and the data to be monitored may change over a period of time (dynamic nature).

1.1.3 Requirements

From the characteristics listed about we formulate the requirements of a monitoring system to support this class of applications. A monitoring system must possess or exhibit the following:

- **A monitoring interface:** Monitoring clients would provide an abstraction to delineate a data subset of interest and specify the set of clients' changes to bemonitored.
- **Generalizability:** Any database that the monitoring system can communicate with may be supported.
- **Transparency:** The monitoring system would be transparent to, and have minimal impact on, the processing capabilities of interactive clients.
- **Timely notification:** The latency between the actual change and its notification to the client should be minimal.

- **Efficiency:** The overhead of computation within the monitoring system and communication to and from the database server must be kept to a minimum.
- **Scalability:** Multiple monitoring clients must be able to effectively monitor multiple clients and relations.

1.2 Current approaches and limitations

Existing systems seeking to provide this capability employ several different methods to observe these changes. One solution that is typically used by applications today is to use the *inherent trigger mechanisms* supported by databases to monitor all changes to a database table and add any relevant results to auxiliary tables. Although it is possible to introduce monitoring behavior within a database using this triggering mechanism, there is no satisfactory means, currently, to send notifications from the underlying system to any external application that requires it. The results of a trigger execution goes back to the user (or login) who initiated the request and not to anyone else. In order for others to observe this change, the shared database needs to be accessed and checked by others. Clients (other than the initiator of the request) need to resort to polling these auxiliary relations in order to decipher which changes had been applied. This approach is unsatisfactory as it results in wasted network bandwidth from polls that do not reveal changes. Depending on the polling frequency the updates received may be stale. Additionally, the approach is *not transparent* to modifying clients as triggers are set on their behalf and clients must be made aware of this.

Another approach involves modifying the underlying database system to push notifications to applications whenever a change is detected to a monitor's interest set. Since the database has a means to communicate changes to any subscribing applications, notifications can be sent to them in real time. However, this approach is not applicable to any database in the same manner as this requires customized changes to the database imple-

mentation. Auditing mechanisms that check for changes after they have been logged in stable storage comprise another approach to this problem. These approaches are explored and contrasted in this work.

1.3 Active Technology and the ECA Agent

The problems with existing systems have been hinted at in the previous section. The sometimes contradictory requirements of generalizability, efficiency, transparency and timely notification render all of these approaches only partially effective. Considering the requirements in their entirety led us to the belief that a database system had to be able to communicate the occurrence of events caused by DML (data manipulation language) operations and their specific updates to external applications in a reactive manner. Active database systems are the ideal vehicle to deliver such a service. An Active DBMS (ADBMS) is one that can continuously monitor situations (specified to the DBMS) to initiate appropriate actions in response to database updates and the occurrences of particular events automatically. Example applications that use active capability are computer-integrated manufacturing, workflow and process control, hospital etc. In all these applications, an ADBMS monitors the situation continuously and provides appropriate responses based on the behavioral rules specified.

ADBMSs are based on the idea of supporting Event-Condition-Action (ECA) rules. An Event is an atomic occurrence or happening. Events may be primitive or composite in nature. Primitive events correspond directly to changes of state caused by a DML operation. Composite events correspond to some combination of these primitive events. The condition can be a simple or a complex query based on the existing database states or a set of data objects. Actions specify the operations to be performed when an event has occurred and the condition evaluates to true. Once ECA rules are specified declaratively it is the responsibility of the ADBMS to monitor situations and trigger the rules

appropriately. They can be leveraged to provide an abstraction for specifying occurrences of interest that need to be monitored, namely the DML operations of inserts, deletes and updates to database tables.

A number of prototypes of ADBMS systems such as HIPAC [2], Ariel [3], Sentinel [4], Starburst [5] and SAMOS [6] have been developed. Most of these systems use an integrated approach that requires access to the source code of the DBMS. In an effort towards separating the active capability from the database, a mediator-based ECA Agent was implemented by Lijuan Li [7] at the University of Florida to provide ECA functionality to a commercial DBMS (Sybase). The next step was a Generalized ECA Agent that was implemented by Zecong Song [8] at the University of Florida. The effort was complemented by adding active capability to commercial database systems at the University of Texas at Arlington. The support of the ECA Agent was extended to add active capability for Sybase ASE by Ganesh Gopalakrishnan [9], Oracle by Yogesh Arvind [10] and IBM's DB2 by Nellainayagam Subramaniam [11]. The use of the mediated approach provides the following benefits:

1. Transparency: The clients have no knowledge of the existence of the ECA Server.
2. Retains system functionality: None of the existing system functionality is lost. Existing clients use an extended SQL syntax that does not conflict with standard SQL syntax as the mediator interprets commands directed to it.
3. Extensibility: The architecture is flexible and with minor modifications can be tailored to meet the specifications of various applications.
4. Multi database support: The mediated approach supports some of the most popular commercial RDBMS platforms that support the JDBC interface. JDBC technology is an API that provides cross-DBMS connectivity to a wide range of SQL databases and access to other tabular data sources, such as spreadsheets or flat files.

1.4 Contributions

The contributions made in this thesis are as follows:

- We present a solution to the generalized monitoring problem using active capability.
- The design of the ECA Monitor, which is the server side mediator providing monitoring capability, and the complementary extensions to the existing ECA Agent are discussed.
- An exposition of key implementation details of the ECA monitor is provided.

1.5 Organization

The rest of the chapters are organized as follows: Chapter 2 covers some of the approaches of existing systems seeking to incorporate a monitoring and notification mechanism. Chapter 3 will cover an overview of the Active technology in the context of the ECA Agent. Chapter 4 deals with the design and architecture of the extended system. The implementation details of the architecture are set forth in chapter 5. Finally, we summarize the contributions made in this thesis and present some avenues for future exploration in chapter 6.

CHAPTER 2

Related Work

In this chapter we will cover the approaches that were introduced in chapter 1 and comment on their suitability for the monitoring problem. We will present notification within the THOR object relational database, the audit based mechanism for monitoring changes in Entegra (a product by Lumigent), and explore pull-based approaches using triggers and snapshots.

2.1 Integrated approaches Notification in the THOR object relational database

2.1.1 Introduction

THOR [12]is an object-oriented database system, intended for use in heterogeneous distributed environments, that allows for objects to be shared between different client applications. The goal of the THOR system is to provide highly-available and highly reliable storage for objects, while supporting safe sharing of these objects by applications written in the Java programming language.

THOR has an implementation of a notification mechanism that notifies a client application upon a change to a given piece of data. The system attempts to address the issue of providing the client with an updated copy of the modified data in a manner that maintains consistency, promotes the rapid update of stale data, and efficiently makes use of the limited network bandwidth and CPU processing resources in the system.

By notifying the client application of object modification, THOR will able to implement a wait-for-change functionality. Essentially, the client applications will be able to "expect" another application to change the value of one of its object. By providing

a copy of the new object, the application can be assured that its transactions are being performed on valid objects. The overhead of polling the server to check if the object has been modified will no longer have to be undertaken by the client, nor will the network be burdened by a potentially heavy load of polling calls.

2.1.2 Architecture

The THOR environment encapsulates all data in the form of objects, where each object has a unique identifier, as well as its own set of methods that allow access and modifications. The architecture is in the form of a client-server model that keeps persistent state of each object. Any persistent object is stored at the Object Repository (OR), which constitutes the server side of THOR. The OR contains a root object, through which all objects are reachable.

The OR is responsible for checking the validity of any potential transaction, and ultimately committing the transaction if valid. The validity of transactions is determined by an optimistic data consistency protocol. The client side of THOR consists of a Front End (FE), which serves as a local cache for the application that runs on top of it. There is a one to one mapping between FEs and applications. An application's FE contains copies of a subset of the objects stored at the OR. The application accesses objects stored at the FE through its own transactions. A commit of a transaction is requested via the FE, which in turn passes the request on to the OR. The FE and OR communicate through a network connection, over which a set of predefined messages can be sent. The architecture is shown in figure 2.1 :

The modifications of the OR to allow for notification can essentially be broken down into four categories. First the OR requires additional state to identify the set of objects for which each FE wishes to be notified. Secondly, the OR provides a mechanism that allows for its various FEs to identify and request notification of desired objects. Next,

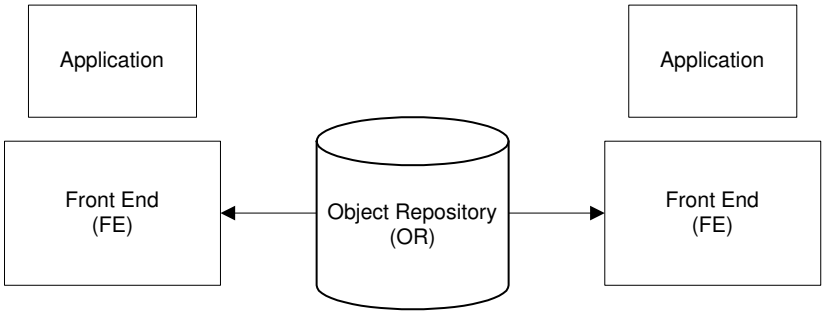

Figure 2.1 Architecture of the THOR notification mechanism.

the notification message is created within the OR that contains information relevant to the notification, including the updated values of the objects. Finally, the OR must implement a method of actually notifying the proper FE's upon receiving a commit that changes the relevant objects.

2.1.3 Shortcomings

THOR satisfies the criteria of efficient and real-time notifications, in a transparent manner. The overheads that it adds to the system are minimal in order to ensure reliable and safe notifications while ensuring that consistency of the data objects is maintained. However, the solution is specific to the THOR object relational database. As the modifications made are specific to the architecture of THOR, it is unlikely that these changes can be effectively translated to other database systems. It is also not clear whether the approach can be satisfactorily extended to any RDBMS.

2.2 Audit Based Approaches Lumigent's Entegra

2.2.1 Introduction

Lumigent Tecnologies' Entegra system [13] was developed as a solution to fulfill the requirements of 21 CFR Part 11 regulation of the FDA [14]. 21 CFR Part 11 specifically

establishes control requirements for electronic records systems and sets out electronic signature requirements for electronic records submitted under requirements of the Federal Food, Drug and Cosmetic Act and the Public Health Service Act. A key requirement for electronic records systems in the purview of 21 CFR Part 11 is the use of computer generated audit trails to document actions that create, modify, or delete electronic records [15]. Data Access Accountability, the capability to determine and document who did what to which data, when, and by what means, is a concept that captures the objectives of the audit requirement under 21 CFR Part 11. The audit trail must capture who made what changes to create, modify or delete electronic records, and when. The technology for generating the audit trail should provide confidence that relevant activities will be captured, that activities will not be inadvertently omitted from the audit trail, and that the audit trail will remain secure. Entegra is an effort by Lumigent to provide a satisfactory vehicle to deliver on the requirements of 21 CFR Part 11 11.10 (e). Minimally Entegra, which is a Data Access Accountability System, must:

- be secure
- be time and date stamped
- record time and date entries and actions that create, modify or delete electronic records, independent of the operator
- ensure that the creation of audit trails does not obscure existing information
- ensure appropriate data retention for the audit trail (at least as long as the underlying electronic record exists)
- ensure availability of audit trails for FDA review and copying.

2.2.2 Design

In this approach, non-trigger audit agents are associated with each database server containing important data. These audit agents are responsible for harvesting information

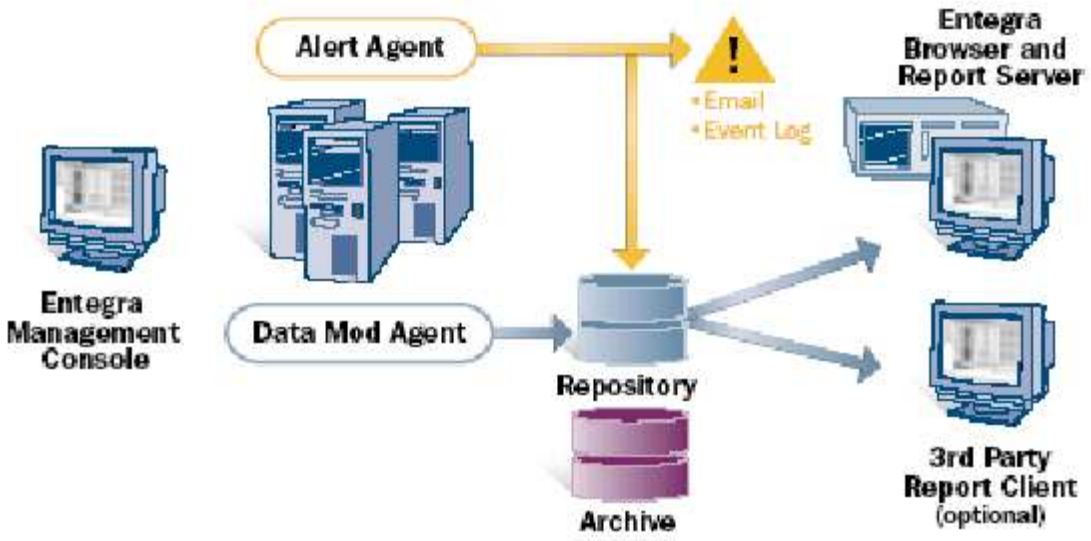

Figure 2.2 Architecture of Lumigent's Entegra.

about data-related activity, and because they operate at the database server, they capture all relevant data activity, regardless of the application used. In addition, applications need not be modified to accommodate this approach. The audit agents harvest information through two primary means:

1. reading the database transaction log, which each database maintains in the normal course of its operation, for data modifications. This does not interfere with the timely execution of transactions, because the analysis can be time-shifted or carried out on machines other than the one hosting the target database.
2. using the database's built-in event notification mechanism to obtain a record of permission changes, schema changes, and data views.

2.2.3 Architecture

The following components appear in a fully configured Entegra deployment [16] as shown in figure 2.2

- **Audit Agent**: Attached to a database server, the audit agent is responsible for harvesting desired information about data modification, data viewing, and structural activity on that server. Together with the Management Console component, this enables Entegra to be configured to capture data for the audit trail as determined by the applicable predicate rule and type of operation relevant to the electronic records.
- **Repository**: The repository receives and stores the information reaped from the audit agent(s). Together with the Archive component, this prevents the audit trail from obscuring existing records and enables copies of the audit trail to be produced for review.
- **Management Console**: The console determines the schedule and configuration of each audit agent for harvesting and transferring information to the repository. Together with the Audit Agent component, this enables Entegra to be configured to capture data for the audit trail as determined by the applicable predicate rule and type of operation relevant to the electronic records.
- **Report server with browser interface**: The report server provides secure mechanisms for processing the repository for query, analysis, and reporting.
- **Archive**: The archive provides long-term storage of access information from multiple repositories. Together with the Repository component, this prevents the audit trail from obscuring existing records and enables copies of audit trails to be produced for FDA review.

2.2.4 Shortcomings

This system meets the criteria of efficient, transparent notification. However, it currently supports only MS SQL server 2000 and 7.0 [17]. It depends on the databases ability to send notifications of changes to views, permissions and schema changes. This

capability does not pervade the gamut of database systems in use today. The audit agent would have to be specifically tailored to acquire the required information from a particular RDBMS and use its application communication primitives in order to extend support that RDBMS. Although this system exerts minimum intrusion on the interactive client by gathering information from log files in stable storage, it does not guarantee real-time notification of updates. This is because database logs are written out periodically to stable storage and this operation introduces a time lag which could potentially lead to bursts of notifications which do not follow the update patterns closely.

2.3 Traditional Trigger based programming

Most modern RDBMSs support the notion of triggered procedures (usually just called triggers in the literature). A trigger is a procedure that is invoked "automatically" on the occurrence of some specified event or trigger condition. Triggers are the traditional approach to efficiently monitor changes in a relational database [18]. They are typically used to ensure referential actions whenever a specific exception is raised (the violation of a specific integrity constraint).

External applications can communicate with the RDBMS to set up triggers which execute on conditions such as the occurrence of a DML operation. Figure **??** shows an Oracle trigger set before a tuple is inserted into the employee table. Any new tuple being inserted, having a null value in the *Manager* field and a *Salary* value greater than 1000, has the *Salary* value updated to 1100:

The complexity of triggers can be considerably greater than this since it has the power of a procedural construct. Exception handling is another feature of triggers. Although, triggers represent a powerful way to specify a condition (the WHEN clause in the example above) and a corresponding action (the statement block defined by BEGIN and END) on the occurrence of an event (BEFORE INSERT in the example above),

```
CREATE OR REPLACE TRIGGER sampletrigger
        BEFORE INSERT ON Employee
        FOR EACH ROW
        WHEN (NEW.Manager is null)
                BEGIN
                        IF (:NEW.Salary > 1000)
                        THEN
                                :NEW.Salary := 1100;
                        END IF;
                END;
```

Figure 2.3 An Oracle Trigger..

their power is limited to actions within a database. The ability to communicate results directly to an external application is limited. For the problem at hand, we need a way to communicate changes to any relation on the events of an INSERT, UPDATE or a DELETE. Although we may, quite easily, write the required results of any of these events to auxiliary tables, we require the application to learn of these results. Currently, there is no uniform way across any given RDBMS that allows this. Applications must resort to some means of polling the auxiliary tables to learn of changes. Network bandwidth is wasted whenever we have polls that have a greater frequency than the updates are made. There are potentially missed notifications whenever applications poll at frequencies lower than the updates are being made.

2.4 Summary

In this chapter we have discussed a number of approaches used for developing a framework to observe changes at the database level. We have described the integrated approach as was taken in the THOR database, an audit-based approach adopted in Lumigent's Entegra and also the traditional trigger based programming approach. Although

they all have their advantages we have seen that none of them completely address the requirements outlined in section 1.1.3. The next chapter provides an overview of the ECA Agent system as a mediator based approach to providing active capability in databases. This will serve as a precursor to our discussion on the ECA monitor: a solution to the generalized monitoring problem.

CHAPTER 3

The ECA Agent

In this section we present an overview of the ECA paradigm and the ECA Agent as a means to provide complete active capability to databases. We seek to provide an understanding of the ECA Agent as a mediator system with an overview of its architecture and a detailed view of some implementation aspects that will be relevant to our later discussion of its extension to provide dynamic monitoring capability.

3.1 The ECA Paradigm

3.1.1 Primitive and Composite events

The Event Condition Action(ECA) paradigm is a powerful notion that can model a very wide and varied set of application scenarios. An event is an occurrence of interest at a specific point in time. **Primitive events** are elementary occurrences and are classified into domain specific (e.g., database), temporal, and explicit events. Database events are associated with the manipulation of data such as update, delete, or insert (on tables) and are executed over a period of time. Temporal events correspond to absolute and relative events that are associated with time. The absolute temporal event is an event associated with an absolute value of time. For example, 4 P.M. on August 15, 1947 is an absolute event. The relative temporal event is an event corresponding to a specific point on the time line, which is an offset from another time point (specified either as absolute or as an event). Explicit (also termed abstract) events are explicitly defined in the application, but their occurrences are either detected outside of the application and conveyed to the

application or the application explicitly raises those events. We are concerned primarily with database events.

A **composite event** is an event that is composed of primitive events and/or other composite events by applying **SNOOP** [19] event operators such as OR, AND, SEQUENCE, NOT. In other words, the constituent events of the composite event can be primitive events and/or composite events.

3.1.2 Parameter Contexts

Four parameter contexts **recent, chronicle, continuous**, and **cumulative** were introduced to provide a mechanism for capturing meaningful application semantics and reduce the space and computation overhead for the detection of composite events. The contexts are defined by using the notions of initiator and terminator for events. An event that initiates the occurrence of a composite event is termed the initiator of the composite event. An event that completes the detection of a composite event is denoted as the terminator of the composite event. For example, a composite event (**E1; E2; E3)**, where ; is a sequence operator, has E1 as initiator and E3 as terminator.

3.1.3 Coupling Modes

Coupling modes were introduced [20] to specify a relative point in time where condition evaluation and action execution should take place after the event is detected, with the constraint that the action will be performed only when the condition is satisfied. There are three coupling modes:

- **Immediate:** When an event is detected, the transaction is suspended, and the condition associated with the event is evaluated immediately. If the condition evaluates to true, the action is executed. The execution of the triggering transaction is suspended while the condition evaluation and action execution are completed.

- **Deferred:** The triggering transaction is continued after an event is detected. Condition evaluation and action execution are performed at the end of the triggering transaction before it commits.
- **Detached (or decoupled)**: Condition evaluation and action execution are carried out in a transaction (or triggered transaction) separate from the triggering transaction. The detached mode can be classified into two types, namely, totally independent and causally dependent. When two transactions are totally independent, the triggered transaction is executed regardless of whether the triggering transaction commits or aborts. On the other hand, the triggered transaction can commit only after the triggering transaction commits for the causally dependent mode.

3.1.4 ECA Rules

An ECA rule is a specification of an event occurrence (primitive or composite as specified by a SNOOP expression), the condition to evaluate on the occurrence of the event and the action to be taken. The condition and action may be specified in any different number of ways. The most relevant example to this discussion would be the mapping of triggered procedures in databases to the ECA paradigm. An occurrence such as a DML operation like an insert, delete or update command would map to a primitive event. The condition is that specified for the trigger to execute the body, which corresponds to the action taken. More complex combinations of these primitive events may be composed using the **SNOOP** language mentioned earlier.

3.2 A mediator approach to active databases

The ECA Agent is a stand-alone system that can, in general, be used with any kind of relational DBMS such as, Oracle or Sybase, and provide complete active capability to

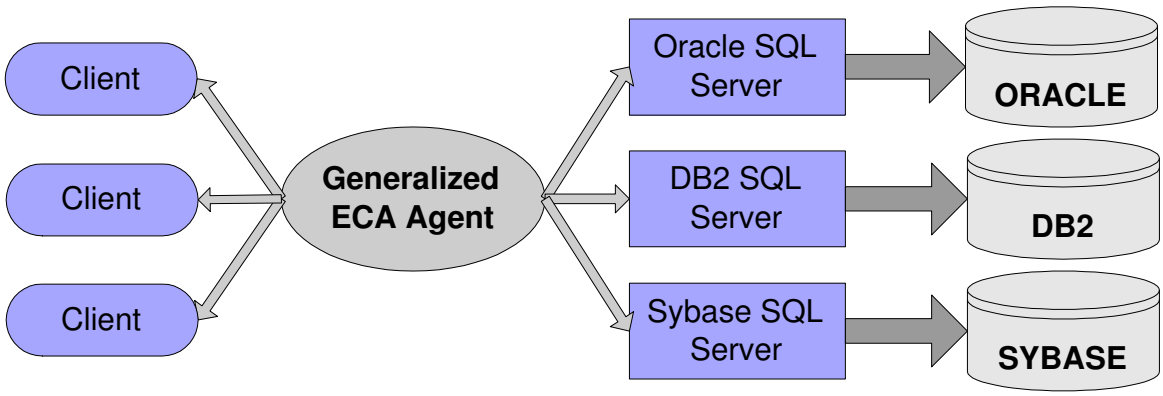

Figure 3.1 ECA Agent - A multi-client, multi-database mediator.

the DBMS. The ECA Agent (a server which is also referred to as the ECA Server) resides between the interactive client(s) and the SQL server(s). Messages from clients are routed to the SQL server through the ECA Agent. Results from the SQL server are returned to the client via the same ECA Agent. The ECA Agent is a multi-threaded program. It can process requests from multiple clients and route them to the appropriate server. In this respect it follows a mediated approach as shown in figure **??** below:

From the perspective of the user, the ECA Agent works just like a virtual SQL server. The presence of the ECA Agent is totally transparent to the client. The ECA server augments the SQL Server with Active capability using the basic trigger capability provided by the SQL server.

We now explore how the ECA Agent can be used to extend a database's native trigger capability and applied in a real world application. Consider a Weather forecast scenario where the following tables WEATHER and WINDSPEED are maintained in our database.

WEATHER: This table stores information about the temperature of a particular city at a certain point of time as shown in Table 3.1

Table 3.1 Weather table (WEATHER)

CITY	TIME	TEM

WINDSPEED: This table stores the wind speed of a city at a particular point of time and its structure is shown in Table 3.2

Table 3.2 Wind Speed table (WSPEED)

CITY	TIME	WSPEED

Now, consider a third table, WEATHERFORECAST. Every time the WINDSPEED table is modified after a modification of the WEATHER table, this table collects temperature information from WEATHER and the corresponding wind speed from WINDSPEED. Based on the values collected, it must decide whether the weather is favorable for people to venture out of their homes. The structure of the WEATHERFORECAST table is shown in table 3.3:

3.2.1 RDBMS Limitations

Every conventional RDBMS implements the same basic functionality. The specific syntax and modus operandi, of course, differ. With regard to implementation details which are necessary to make our discussion concrete, we will restrict our discussion to the Sybase Adaptive Server Enterprise 12.0 system (henceforth simply referred to as Sybase).

Table 3.3 Weather Forecast table (WEATHERFORECAST)

CITY	TIME	TEM	WSPEED	FAVOURABLE

It is also relevant as the system has been implemented for Sybase. Sybase provides us with the ability to create triggers on INSERT/DELETE/UPDATE operations. However, in the example at hand, we need to monitor a modification of WEATHER followed by a modification of WINDSPEED. Such a requirement cannot be specified and is not directly supported in Sybase (or other DBMSs). However, this may easily be constructed as a SEQ(sequence) operation in the SNOOP language. This drawback of an RDBMS, namely, the inability to monitor a combination of primitive operations (on the same or different table) and take appropriate actions, is overcome by using the ECA Agent. In addition to this limitation, Sybase does not support prioritized firing of triggers, nor does it support the notion of detecting events in different contexts. Also, particularly in Sybase, there is a restriction that only one trigger may be defined for each DML operation on a table. Thus, we may have at most 3 triggers (one each for the INSERT, DELETE and UPDATE operations) on any given table. The use of the ECA Agent allows us to bypass these limitations.

3.2.2 Capabilities added by the ECA Agent

With the help of the ECA Agent, we associate each DML operation to a unique event name (primitive event). Using the Local Event Detector (LED), the ECA Agent creates a node for these primitive events. Now, if the user wants to monitor a combination of these primitive events, then he/she needs to specify a composite event trigger. This results in a composite event node being created inside the LED. Whenever a DML operation occurs, if the user has created a primitive event on that DML operation, a trigger is fired following which, the ECA Agent notifies the LED about the occurrence of the primitive event. The LED registers the occurrences of this primitive event and propagates the same to the composite event node. The LED, as part of its duties, checks if the occurrences of these primitive events satisfy the conditions for the composite event to

be detected. Using the Java LED, composite events can be detected in different contexts and coupling modes. If it is satisfied, then the LED detects the occurrence of a composite event. Following this, the action part specified in the composite event trigger is executed based on priority. Next, we describe the design of the ECA Agent, the different ECA Agent components, their functionality and how they interact with one another. We note the following characteristics of the ECA Agent:

- The ECA Agent can associate RDBMS operations with primitive events. It is able to detect primitive events, and supports composite event detection, thus extending the native trigger capability of Sybase.
- Multiple users may interact with the ECAAgent and hence with the SQL server.
- Since Sybase does not support the notion of events, information about the events being created for a particular operation in the trigger statement must be retained. In order to do this, the SQL trigger syntax provided by the vendor (Sybase) is extended to include this data in the trigger statement.
- The ECA Agent has a module that is responsible for event information extraction. It does this by embedding specific event creation information in the SQL statement which is interpreted to create the corresponding events in the LED.
- Information about the creation of events is persisted. This allows reconstruction of the events created by users between sessions or crashes.
- The dropping of events and triggers is supported seamlessly to allow native triggers and extended triggers to reside concomitantly within the Sybase Server.

3.2.3 Sybase Triggers

A trigger is a stored procedure that goes into effect when you insert, delete, or update data in a table. You can use triggers to perform a number of automatic actions, such as cascading changes through related tables, enforcing column restrictions, compar-

ing the results of data modifications, and maintaining the referential integrity of data across a database [21].

Adaptive Server imposes several restrictions on the use of triggers and their capabilities [21]. Some of the most relevant limitations are:

- A table can have a maximum of three triggers: one update trigger, one insert trigger, and one delete trigger.
- Each trigger can apply to only one table. However, a single trigger can apply to all three user actions: update, insert, and delete.
- You cannot create a trigger on a view or on a temporary table, though triggers can reference views or temporary tables.
- Although a truncate table statement is, in effect, like a delete without a where clause, because it removes all rows, it cannot fire a trigger, because individual row deletions are not logged.
- You cannot create triggers on system tables. If you try to create a trigger on a system table, Adaptive Server returns an error message and cancels the trigger.

The use of the ECA Agent overcomes the first restriction as it is able to combine trigger bodies and procedures so that they execute consecutively within a single trigger.

3.3 Architecture

In order to understand how the ECA Agent is extended to support monitoring capability we cover a brief overview of some of the important architectural components of the ECA Agent. Components relevant to our discussion are described in greater detail in the following subsections. Figure shown below represents the ECA Agent and a single interactive ECA client.

3.3.1 ServeOneClient

This module accepts connection requests and input from multiple clients, spawning a new thread for each new client. The task of setting up an environment or reconstructing one from a previous session is this module's responsibility. Part of this setup involves creating a unique system wide directory structure for each individual user. This is done in order to prevent clashes between files (generated for registering events) created by different users and thus allow multi-user capability. It also invokes the Persistence Manager to reconstruct any event graphs that may have existed in a previous session. Commands sent from the client are routed through to the Language Filter. Once the command is executed a check is made to see if any events were raised by checking that clients **NOTIFY** table, which is described later. Once this is done, the results, which may include the raising of primitive and composite events is transmitted back to the interactive SQL client.

3.3.2 Language Filter

Both native and extended syntax commands are routed through the Language Filter. Client requests such as the SELECT statement and the normal CREATE trigger commands are considered pure SQL commands. These are sent to the SQL server via the Java Database Connectivity (JDBC) module. Requests that include the creation of a primitive event, a composite event (trigger), or the dropping of a trigger (created on primitive or composite events) are considered ECA commands. An ECA command is passed to other modules of the ECA Agent for further processing. The ECA trigger command given by the user can fall into one of the following four categories. Italicized words indicate the extensions for including event information.

1. *Primitive event triggers*: are triggers created on one of the primitive operations such as Insert, Delete or Update. We can associate a primitive event for each of

```
CREATE trigger trigger_name event event_name on [ owner.] table_name
        for {INSERT , UPDATE , DELETE}
        as SQL_statements

Or, using the if update clause:

CREATE trigger trigger_name event event_name on [ owner.] table_name
        for {INSERT, UPDATE}
        as
                [if UPDATE( column_name)
                [{and | or} UPDATE( column_name)]...]
                Begin
                        SQL_statements
                End
                [if UPDATE( column_name)
                [{and | or} UPDATE( column_name)]….]
                Begin
                        SQL_statements
                End…..
```

Figure 3.2 Sybase Primitive Event Trigger Syntax.

the different types of triggers. The syntax for setting primitive event triggers in Sybase is shown in figure

2. *Repeat triggers on a primitive event*: Once a primitive event has been created for a particular operation, the rest of the triggers that the user wants to create on the same operation and also associate with the same primitive event are called as repeat primitive triggers. The syntax for setting repeat primitive event triggers in Sybase is shown in figure 3.3
3. *Composite Event triggers*: These triggers are created based on the occurrence of one or more primitive (or composite) events. The syntax for setting composite event triggers in Sybase is shown in figure 3.4

CREATE trigger *trigger_name* ***event event_name*** = *snoop expression* :
[*coupling mode*] [*parameter context*] [*priority*]
Begin [*sp*]
SQL_statements
End

snoop expression - Expression following the syntax and semantics of the SNOOP event definition language - .

coupling mode - Immediate, deferred, detached
p*arameter context* - Recent, continuous, cumulative, chronicle
priority - any positive integer.

sp - stored procedure if the user wants the SQL statements to be performed as a transaction.

Figure 3.3 Sybase Repeat Primitive Event Trigger Syntax.

Create trigger *trigger_name* ***event event_name*** = ***snoop expression*** :
[***coupling mode***] [***parameter context***] [***priority***]
Begin [***sp***]
SQL_statements
End

snoop expression - Expression following the syntax and semantics of the SNOOP event definition language.

coupling mode - Immediate, deferred, detached
parameter context - Recent, continuous, cumulative, chronicle
priority - any positive integer.

sp - stored procedure if the user wants the SQL statements to be performed as a transaction.

Figure 3.4 Sybase Composite Event Trigger Syntax.

4. *Repeat triggers created on composite events:* The Java LED allows several rules (combination of condition and action based on certain semantics) to be associated with a single event node. Consequently, the user can specify several action portions that get fired when a composite event is detected. In order to have several rules associated with the node, the user would have to create repeat composite event triggers.

Based on the category of the ECA command, the Language Filter routes these commands to the appropriate module. Similarly, if the client request is a "Drop Trigger" command, it is left to the language filter to figure out if the trigger being dropped is a primitive trigger or a composite trigger and route them to the proper modules.

3.3.3 ECA Parser

The extended trigger syntax allows DML operations in a RDBMS to be associated with events in the Active Technology framework. If the user wants to monitor database operations, triggers are created. These triggers fire whenever DML operations (such as Insert/Update/Delete) occur. In order to associate the operations with events, an extended SQL syntax for creating triggers on DML operations with associated events is used. When this modified SQL command is sent to the ECA Agent, the ECA parser extracts the event information and creates the corresponding events in the LED. An API to create primitive, repeat primitive, composite and repeat composite events is provided. Since information about a composite event is embedded within the extended SQL syntax the relevant composite event information, including the SNOOP expression, the parameter context, coupling mode and action string need to be extracted. The ECA Parser uses the AnalysisCompositeEvent API to perform this task.

3.3.4 Persistence Manager

Since the LED is a runtime (main memory) structure used to create and detect events, the event graph is valid only for that session. Once the user logs out (or the server or the client crashes), events and triggers should be re-created when the user logs in again based upon what has been persisted in the system tables. The ECA Agent creates a Java source file which may be compiled and loaded at runtime for registering events with LED. In order to make the appropriate decision, the ECA Agent should have access to information about the triggers and events that were created by the user during the previous session. The Persistence Manager is responsible for the task of persisting event information into the database. It makes use of System tables for storing ECA information associated with a trigger/event. It uses the following System tables for persisting the event information.

1. SYSPRIMITIVEEVENT: This table stores information about the primitive events. Before the trigger from the user is submitted to the SQL server, it is parsed, analyzed, and transformed (if necessary), so that, inside the SQL server, it is persisted as a trigger acceptable to the SQL server and capable of performing the ECA actions, when the trigger is fired. In the table SYSPRIMITIVEEVENT, the first two columns, i.e., the DBNAME and USERNAME, correspond to the name of the database (i.e., the instance of database to which your account has access) and the login id of the user respectively. The column BEAFOPERATION, which is not used in Sybase , stores information about whether a trigger is a BEFORE event trigger or an AFTER event trigger. TIMESTAMP stores the time of creation of the trigger while VNO stores the number of occurrences of the event up to that point (it is initialized to 0). The contents of the table SYSPRIMITIVEEVENT are used for the following purpose:

Table 3.4 SYSPRIMTIVEEVENT table

DBNAME	USERNAME	EVENTNAME	TABLENAME
OPERATION	BEAFOPERATION	TIMESTAMP	VNO

- The column eventname is used to authenticate the name of the primitive event. The name of the event has to be unique. So, when the user tries to create a new event, the ECA Agent checks this table to see if the event name specified by the user is a duplicate event name.
- The column version holds the version number of the latest occurrence of a primitive event.
- Inside the database, the table associated with a particular user is identified by concatenating the name of the database, the user ID and the name of the table.

It has been illustrated in Table 3.4.

2. SYSCOMPOSITEVENT: stores information about the composite events.
3. SYSECATRIGGER: The SYSECATRIGGER table is used to store information about all the ECA triggers, both primitive and composite. This table is primarily used for making sure that the trigger names are unique. The structure of SYSECATRIGGER is shown in Table 3.5.

Table 3.5 SYSECATRIGGER table

DBNAME	USERNAME	TRIGGERNAME	TRIGGERPROC	TIMESTAMP	EVENTNAME

The Persistence manager also performs the task of creating triggers. The extended SQL syntax described earlier cannot directly be used to create triggers. In order to

overcome this problem, the Persistence manager extracts the event related information from the SQL command. Once this is done, the ECA Agent appends certain ECA actions to the trigger body. These ECA actions primarily update system tables with the occurrence of the primitive event and collect parameters needed for composite event detection. This step is necessary since composite events are detected outside the scope of the SQL Server. Thus, when a trigger is fired, the ECA Agent raises the corresponding primitive event that it has detected in the ServeOneClient. It detects this by checking the **NOTIFY** table described later. The Java LED API is used with the appropriate calls and parameters to raise the event once it has been detected. These parameters are the event name and table name on which the trigger was defined and the version number i.e. the global number that corresponds to the occurrence of that primitive event.

In addition to these parameters, the ECA Agent should also collect the transient values (within Sybase they are the deleted and inserted tables) that exist after a trigger is fired. The reason we need to collect these transient values is that they cannot be accessed outside the body of a trigger statement. Whenever a composite event is detected, we need to access the parameters that existed when the trigger(s) fired (i.e., when the primitive event(s) was detected), so that the action portion of the composite event trigger acts upon the proper set of values. For the composite event trigger to have access to transient values that existed as a result of the firing of a primitive event trigger, the ECA Agent creates tables that can hold these transient values along with the occurrence number of the primitive events. We explain these tables in a later section. The Persistence manager uses the following system tables for updating the information when trigger is fired.

1. **VERSION**: This table stores the version number of the occurrence of a primitive event. It is updated globally; that is, irrespective of the type of primitive event that is raised; the Version table gets incremented by one each time. Table 3.6shows the

structure and content of Version after three primitive event occurrences have taken place.

Table 3.6 VERSION table

VNO
3

2. **NOTIFY**: This table stores the parameters associated with a primitive event occurrence. Every time a trigger defined on a primitive event is fired, it populates this table. Once this table is populated, the ECA Agent fetches the values, and notifies the LED about the firing of the trigger. The LED then realizes that a primitive event has been raised and proceeds to register the same. Table 3.7 shows the contents of Notify after a trigger defined on a primitive event is fired.

Table 3.7 NOTIFY table

EVENTNAME	TABLENAME	VNO

Once the ECA Action portion and the user action portion are appended to the trigger statements, the trigger is created in the database. The Persistence manager is also invoked on certain other occasions. For example, If the ECA Agent or the SQL server crashes and the user starts a new session the ServeOneClient module invokes the Persistence manager to re-create the events, rules and resets the system tables to the appropriate values.

3.3.5 Drop Trigger

Within Sybase, the ECA actions that need to be performed when a trigger fires and the actions specified by the user are cleanly separated. This module is responsible for dropping an event and the associated trigger. It has two sub modules:

DropPrimTrigger - is a sub module under drop trigger module that is used for dropping the primitive event and its associated trigger

DropCompTrigger - is a sub module under drop trigger module that is used for dropping the specified composite event, the associated rules and the associated class files.

It is important that an event(primitive or composite) not be dropped if a composite event is subscribed to it. In order to keep track of this information the SYSDROP table is maintained and its structure is shown in table 3.8.

Table 3.8 SYSDROP table

CONSEVENTNAME	CONTEXT	COMPEVENTNAME

CONSEVENTNAME - constituent event name

CONTEXT - the context on which the composite event is defined

COMPEVENTNAME - composite event name DropPrimTrigger

3.3.6 Java Local Event Detector

A relational DBMS can only detect primitive events, such as Insert, Update and Delete. In order to enhance the Active capability, we use Java LED so that we can augment the RDBMSs with the capability to detect composite events and execute the rules associated with these events. The LED when used in a stand-alone mode detects

both primitive as well as composite events. A primitive event can be either a method call from an application (for non-database applications) or a database update operation. However, when used with the ECA agent, the primitive event detected by the SQL server as the execution of a trigger needs to be conveyed to the ECA agent so that it can invoke the corresponding primitive event on the LED. Composite events can be a combination of these primitive events (or other composite events). The composite events can be detected in one of the four contexts based on how the user wants to detect it.

3.3.7 Transient table value access external to a trigger body

ECA Agent creates certain tables for each relation on which the user has created a primitive event. These tables are of the form **R_Inserted**, **R_Inserted_tmp**, **R_Deleted** and **R_Deleted_tmp**, where R refers to the name of the relation. Of these four tables, the tables of the form _Inserted and _Deleted are populated with transient values of a relation that existed at the time the trigger was fired and a primitive event occurrence is detected.

The necessity for the ECA Agent to collect these parameters is as follows: The transient values that exist when a trigger fires are accessible only from within the trigger body. However, these values need to be accessed at a later time when a composite event is detected, so that the proper parameters associated with the constituent primitive event is available in the action portion of the composite event. In order to get the proper occurrence of the primitive event, we need to separate the values that exist in the **R_Inserted** and **R_Deleted** tables. Therefore a fifth column, which consists of the version number of the occurrence of that primitive event is introduced.

The tables of the form _tmp are known as transient tables. These tables are created at the time of creating **R_Inserted/R_Deleted** tables. They contain the transient values that existed in a table when the primitive events that were created on that table

were detected. These tables are populated when a composite event is detected. If the user is creating an event on either Insert or Update, then the ECA Agent creates **R_Inserted** and **R_Inserted_tmp** tables (along with **R_Deleted** and **R_Deleted_tmp** for update as it is a delete followed by an insert and hence has values in both tables). If the user is creating an event on Delete operation, then the ECA Agent will create a **R_Deleted** and **R_Deleted_tmp** tables. The **R_Inserted** and **R_Deleted** tables will have all the columns of the table on which the primitive event has been created.

In addition to this, it will have a column for storing the Version number of the occurrence of the primitive event. For example, if the user has created a primitive event trigger on the table Weather for Insert operation, then the structure of the table **R_Inserted** will be as shown in Table 3.9. The contents of the Transient tables will be the result of a join (based on Version number) between the **R_Inserted** table and Version table.

CITY	TIME	TEMP	WSPEED	VNO

Table 3.9 R_Inserted structure

3.4 Module Interaction for Event Creation

In this section, the different stages of processing that a client request undergoes after being submitted to the ECA Agent are covered. All interactive client requests are routed through the ECA Agent before being sent to the SQL server. After a client logs on to the ECA Agent and submits a request, the ECA Agent listener (ServeOneClient) sends this request to the Language Filter. The Language Filter checks to see if the request is a normal SQL command or an ECA command. If the client request is a normal SQL

command, the Language Filter sends it to the SQL server directly by making a JDBC call. On the other hand, if it is an ECA command, then it can fall into one of the following categories:

1. A trigger being created on a primitive event or a repeat trigger on a primitive event.
2. A trigger being created on a composite event or a repeat trigger on a composite event.
3. A drop trigger command for dropping either a primitive or a composite event trigger.

For the first two cases, control is routed to the ECA Parser in order to create the appropriate events. In the third case the Drop Trigger module is invoked.

We will explain the interaction between the modules by taking an example of creating a primitive event trigger and a composite event trigger. Assuming that the user command is for creating a trigger on a primitive event, the Primitive Event Parser first validates the trigger syntax, the event name and the trigger name. It extracts the name of the event and creates a ".java" file containing a method to registering the event with the LED. The naming convention for the file includes the name of the event. This file is then placed in a system wide unique directory structure having the format ***"username_database"***. The file is compiled, loaded and the event registration method invoked to set up the LED thus creating a primitive event node. Following this, control is passed on to the Persistence Manager module wherein event information is persisted in the system tables and the trigger is created in the SQL server. If the user now performs an operation on which the primitive event has been created, the corresponding trigger is fired and it populates the NOTIFY table with the parameters associated with the occurrence of primitive event. Control returns to the ECA Agent which checks the NOTFIY table. If tuples exist in the table, the appropriate events are raised within the LED. The LED propagates the event to all composite event nodes that have the primitive event as a

constituent event. This completes the process of creating and detecting a primitive event.

3.5 Summary

In this chapter we discussed the architecture of the ECA Agent and explored some of the relevant implementation details that have a bearing on the design of the monitoring system. The next chapter will outline the design of the monitoring system in the context of the ECA Agent's mediated approach. The modules introduced in this chapter are an integral part of architecture of the ECA monitor and their extensions will be addressed in the next chapter.

CHAPTER 4

Design

In this chapter we discuss the design of a mediated monitoring framework and apply it to the Sybase ASE RDBMS. This framework may easily be extended to provide active monitor capability to any one of the RDBMSs that the existing ECA Agent supports as we employ generic facilities available in all RDBMSs. At the time of this writing, the RDBMSs that are supported are Oracle 8i, IBM's DB2 7.0 and Sybase ASE 12.0. In the previous chapter we visited the overall architecture of the ECA Agent and surmised that it would be a viable platform on which to build our monitoring abstraction. To support our claim, we discuss the requirements described in section 1.1.3 and how the ECA Agent can be made to fulfill them. Furthermore, we provide the framework through which we intend to supply this capability and the overall design of the monitoring agent.

4.1 Requirements

In this section we recall the requirements of a monitoring abstraction as outlined in section 1.1.3. We augment that discussion as follows:

A monitoring interface Monitoring clients would provide an abstraction to identify a data subset of interest and specify which set of clients' changes the monitoring client wishes to observe. The monitor must be able to dynamically add monitors of "interest" i.e., the relations of to be observed, the interactive clients modifying it (with regard to this relation), and a means of selecting a subset of this relation. By dynamic, it is implied that "interests" may be added, deleted and modified at runtime.

Generalizability Any database that the monitoring system can communicate with may be supported. In pursuit of this goal we must take into account that we may make use of only a common set of facilities supported by the RDBMSs we wish to support. This was also a prime objective of the ECA Agent's design and the only conditions that are required of an RDBMS are :

- It must have minimal native trigger support (along the lines of Sybase) where triggers may be added and dropped on the primary DML operations of insertion, deletion and updation.
- All communication between the database server and the ECA Agent takes place using a common interface, namely Java Database Connectivity (JDBC).

Transparency: The monitoring system would be transparent to and have minimal impact on the processing capabilities of interactive clients. Our objective of transparency requires that the existing interactive clients remain blissfully unaware of the actions taken by the monitoring system. Interactive clients must be able to issue all commands that an ECA Agent client may legally issue under normal conditions (where no monitoring activity is present). This implies that any action taken on behalf of the interactive client must not limit the interactive client's ability to create and drop triggers, primitive or composite events or any other RDBMS supported objects. In addition, the communication of any auxiliary results pertaining to monitoring activities should be ***available strictly to monitoring clients only***. Interactive clients must be shielded from any data extraneous to its operations.

Timely notification: The *latency* between the actual change and it's notification to the client should be *minimal*. The ECA Agent allows real time notification of primitive and composite events by extending the native trigger capabilities of supported

RDBMSs. It is feasible to use this to our advantage and add a small footprint to support active monitoring.

Efficiency The overheads of computation within the monitoring system and communication to and from the database server must be kept to a minimum. A minor degradation in performance will have to be tolerated by the clients, as in any process of observation this is unavoidable.

Scalability Multiple monitoring clients must be able to effectively monitor multiple clients and relations. The ECA Agent is a multi-threaded, multi client server which has been designed for extensibility. In this regard it would make the ideal platform for a scalable solution to our problem.

4.2 Leveraging the mediated approach

As can be seen from the previous section, the ECA Agent is a viable platform using which we can build the monitoring system that satisfies the requirements mentioned earlier. To maintain generalizability we made the conscious decision to integrate monitoring capability within the mediator. This approach is shown in figure 4.1.

Since we wish to monitor changes to a relation, the simplest answer would be to extract those tuples that are changed during any DML operation. The inserted and deleted tuples would reside in the temporary "inserted" and "deleted" tables of each table in a database when undergoing any updation. Although this approach has shortcomings, which have been discussed in section 2.3, it is the only way general way in which one can actively obtain information of a DML operation. It allows us to identify the user making changes as well as the changes themselves.

The ECA Agent maps primitive events to triggers. It is in this fashion that it can actively receive notifications of DML operation occurrences. The primitive event trigger body simply inserts parameters relevant to the detection of a primitive event (described

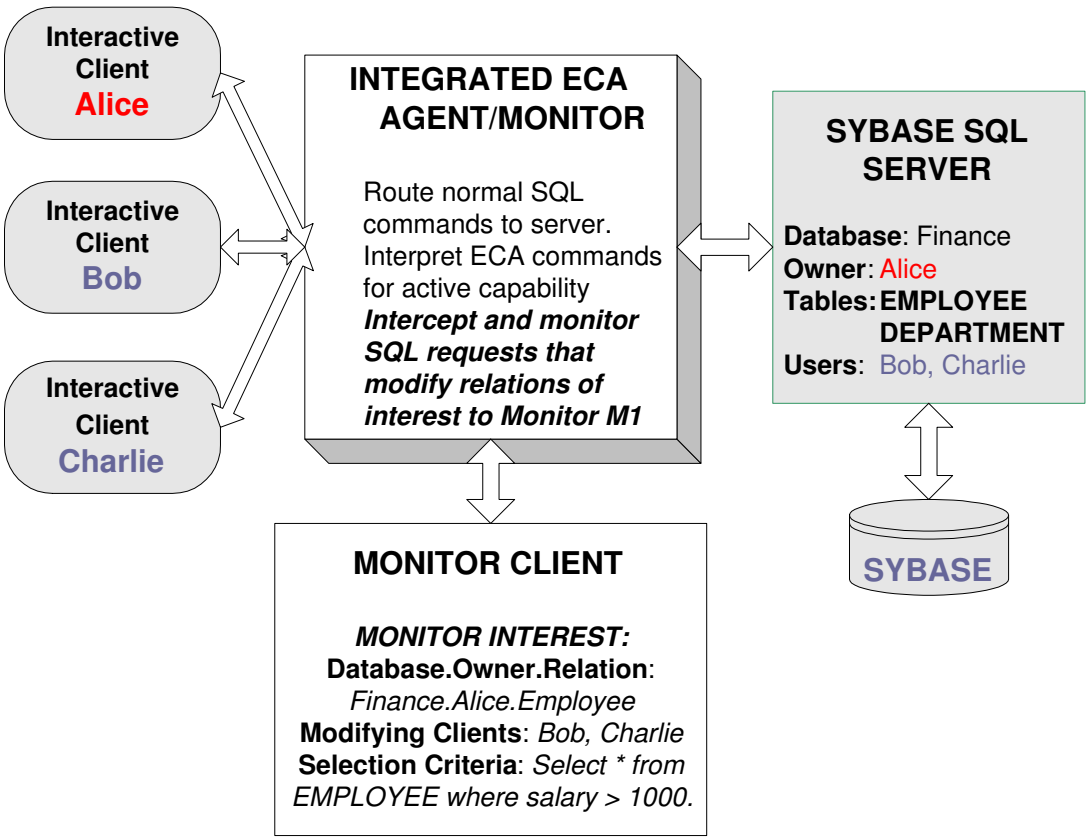

Figure 4.1 Scenario with the integrated ECA Agent/Monitor.

in 3.3.4). These parameters identify the *table, event* and version *number*of the triggered event and are inserted by the actions specified in the trigger body. The ECA agent simply polls this table, determines that a primitive event has to be fired and proceeds to do so with the Java LED. The LED, which has been setup to execute ECA rules carries processing further.

Our approach uses the same method insofar as detection of primitive events is concerned. The key idea underlying our rationale is: by ensuring that a primitive event is set on all the DML operations for a table of interest to a monitor, we can be notified of any changes to that table. Thus we require 3 primitive event triggers to be set, one each for the INSERT, UPDATE and DELETE operations. The monitor would detect

these changes, determine whether the modifying client was part of any monitor's interest, extract the relevant tuples if it was of interest, and actively propagate these results to the client. The following factors have to be considered:

- If the owner has already inserted a primitive event trigger then we may extract the relevant information and propagate it to the ECA monitor. However, if a primitive event trigger has not been installed for a particular DML operation we must explicitly set it on behalf of the owner.
- To ensure transparency, these auxiliary events **must not** be detected by the interactive client.
- In Sybase, the limitation of one trigger per DML operation, makes it incumbent on the monitor to drop the auxiliary event whenever the client wishes to create a primitive event on a given DML operation. This is done so that the clients operations are unhindered. Since the primitive event the client creates will also suffice for the observations on that particular DML operation, the monitoring process can proceed normally.
- Whenever a client drops a primitive event, the monitor must detect this condition and insert an auxiliary event in its place so that monitoring may continue.

4.2.1 Notification and retrieval of changes

We now show how the proposed architecture would operate with an illustration. Consider the following scenario: Alice sets a primitive event trigger on the EMPLOYEE table for the INSERT operation that removes tuples into the table having salaries less than the stipulated minimum for the corresponding department. The trigger is shown in figure 4.3.

Monitor M1 now wishes to monitor all changes to this table made by the users Bob and Charlie. Since the monitor shares the address space with the ECA Agent it

```
CREATE trigger AliceInsertTrig event AliceInsertEvent on Alice.EMPLOYEE
   for INSERT as
   if
   (SELECT count(*) from Department, inserted
         where Department.id = EMPLOYEE.DeptID
               and Department.MinSalary > EMPLOYEE.Salary) > 0

   begin
         delete * from EMPLOYEE, Department, inserted
         where EMPLOYEE.DeptID = Department.DeptID and
               Department.MinSalary > EMPLOYEE.Salary and
         EMPLOYEE.ID = inserted.ID
         print "Employee salary below Department minimum."
   end
```

Figure 4.2 Insert Trigger set by owner of EMPLOYEE relation.

has access to all interactive clients that are sending SQL commands. The monitor may intercept these SQL commands but may only decipher whether they have the potential to update the **EMPLOYEE** relation. In particular, the Language filter module will identify DML operations and extract the fully qualified relation which the SQL statement is modifying. This information will be used to determine whether or not to check for changes to a monitor interest. To confirm these updates the monitor must set a primitive event trigger that will capture the users updates. A monitor must be explicitly given the authority (access to authentication information such as the owner's name and password) to set triggers on the relation. Primitive event triggers will be set on the update and delete operations and the trigger body will follow the most basic syntax of a primitive event trigger. The trigger is interpreted by the ECA Parser and the final syntax supplied to Sybase is as shown in figure **??**

Note that the entries in the inserted and deleted tables are maintained in the **EMPLOYEE_inserted**, **EMPLOYEE_inserted_temp**, **EMPLOYEE_deleted**, and **EM-**

```
Create trigger BASICTRIGGER on PRIMEVENT for INSERT as
Begin
        update SYSPRIMITIVEEVENT set VNO=VNO+1
        where EVENTNAME='EV_MON_INSERT'
        update VERSION set VNO=VNO+1
        insert into EMPLOYEE_inserted select inserted.ID,
                inserted.Name, inserted.DeptID, inserted.Salary,
                VERSION.VNO from inserted, VERSION
        insert into NOTIFY select EVENTNAME, TABLENAME,
                VERSION.VNO, user_name(), db_name()
                from SYSPRIMITIVEEVENT, VERSION
                where EVENTNAME='EV_MON_INSERT'
        delete from EMPLOYEE_inserted_tmp
        insert into EMPLOYEE_inserted_temp select inserted.ID,
                inserted.Name, inserted.DeptId, inserted.Salary from inserted
        delete from EMPLOYEE_deleted_temp
        insert into EMPLOYEE_deleted_temp select deleted.ID,
                deleted.Name, deleted.DeptID, deleted.Salary from deleted
        exec MON_EMPLOYEE_DELETE_PROC
End
```

Figure 4.3 Monitor's Delete Trigger set on Alice's behalf.

PLOYEE _deleted_temp tables as mentioned in section 3.3.7. Also, observe that the information inserted into the **NOTIFY** table now includes username and database information of the modifying entity. The triggers must record the identity of the modifying client, and the current database, to allow the monitor to correctly send notifications to the interested monitors.

There are now three events: *AliceInsertEvent*, **EV_MON_DELETE** and **EV_MON_UPDATE** on the EMPLOYEE table. Whenever any DML operation is performed on this relation the appropriate trigger associated with one of these events will fire. As part of its execution it will insert notification information within the **NOTIFY** table. On completion, the ECA Agent portion of the integrated ECA Agent/Monitor will select only the information corresponding to *AliceInsertEvent* from the **NOTIFY** table. On the other

hand, the Monitor portion of the system will extract information for all events as it need to monitor all DML operations. This information is then added to internal data structures in the ECA Monitor. Processing, from this point onwards, involves identifying all the monitor clients who are interested in changes to **EMPLOYEE** (in our example it is limited to M1 but there may be multiple monitoring clients with the same interest). Changes to the relation are now retrieved from the **EMPLOYEE_deleted** and **EMPLOYEE_inserted** transient tables. As the schema of these tables incorporates version information, a join between the tuples retrieved from the **NOTIFY** table and the transient tables yields the appropriate changes. This information is propagated back to the appropriate monitoring clients.

In the event that the user Alice wishes to drop the insert trigger she has set the Monitor must step in to immediately set up another trigger in order to avoid any missed updates on that operation. Thus the monitor will set up a trigger **MON_EMPLOYEE_INSERT** under the **EV_MON_INSERT** primitive event. This trigger will simply copy all information from the temporary "inserted" table into the **EMPLOYEE_inserted** and **EMPLOYEE_inserted_temp** tables.

4.3 The ECA Monitor

We now detail the design and its rationale in the following sections. The monitoring interface and the Monitor server's architecture, integrated with the ECA Agent is described. The purpose and design of individual components of the Monitor server are explained.

4.3.1 Monitoring Interface

The Monitor client is a simple interface that has been created to specify monitor interests and visualize the changes corresponding to those interests. The interface allows the specification of a set of monitor interests. Each monitor interest is composed of:

1. **Relation of Interest**: The fully qualified relation which is to be monitored specified as *"DatabaseName.Owner.TableName"*. In our running example, we may specify ***"Finance.Alice.Employee"*** as our relation of interest.
2. **Modifying Clients**: A list of clients authorized to make changes to that relation. These clients are users who are granted access and modification privileges to the given relation. In our example, we may specify *"Bob, Charles"* as the modifiying clients.
3. **Selection Criteria**: In order to specify a subset of the changed tuples we may use the power of SQL predicates to selectively view changes to the relation. In our example we may specify it as:

   "SELECT * from Finance.Alice.Employee where Salary ¿= 1000"

The example above illustrates the monitor's intent to view all inserted, deleted and updated tuples of the relation EMPLOYEE, owned by Alice in the *"Finance"* Database, modified by the interactive clients *"Bob"* and *"Charles"*.

The visualization of these results would flag deleted and inserted tuples as such. Updated records would be represented by two entries: one tuple corresponding to the old (deleted) version and one tuple corresponding to the new(inserted) values. In addition to this, there must be a means of identifying how recent the updates are. Thus, for each tuple, the event version number corresponding to its detection by the ECA Agent is appended (refer section 3.3.4). We may also include time stamp information if available. Temporal changes to data can be observed in this manner.

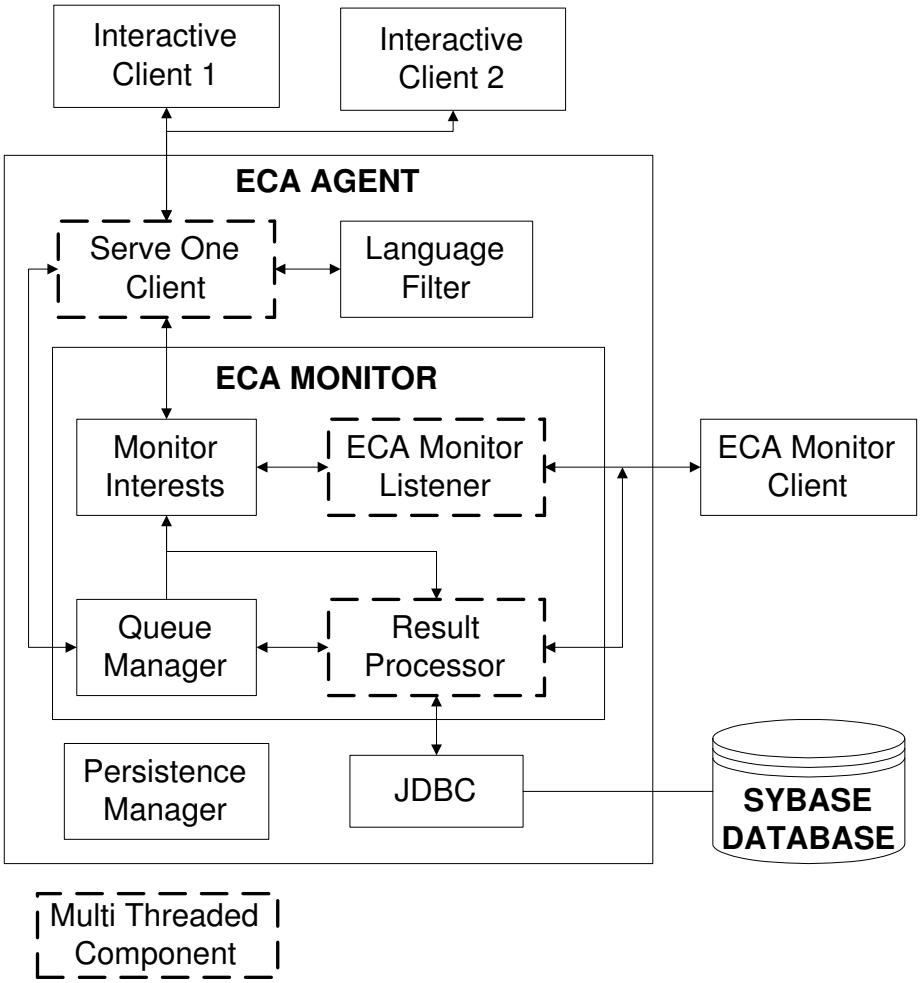

Figure 4.4 Architecture of the integrated ECA Agent/Monitor.

The relation between the Monitor Client and the integrated ECA Agent/Monitor server are shown in figure 4.4. The function and design of the individual modules within the ECA Monitor are described in the sections below.

4.3.2 ECA Monitor Listener

The Monitor Listener is the entry point for all requests from the monitor client. The monitor client may specify a set of monitor interests as specified in the example above. A monitor interest may by added, deleted or modified. Any monitor interest may

be modified with respect to the clients and the selectivity conditions specified for a given table of interest. The functions of this module include the creation, deletion and update data structures that internally represent a monitoring client's interests.

This module is multi threaded and for each unique monitor client that subscribes to the monitor server there will be a dedicated thread spawned to service its requests. The module extracts the monitor interests and sends them to the Monitor Interest module where further processing ensues. All communication with the client takes place via Java TCP/IP sockets. One socket is shared by the listener and result processor pair which handle communication with one monitor client.

4.3.3 Monitor Interest

All monitors and their interests are stored within several shared data structures that need to be properly synchronized and managed. The information stored here for each monitor interest includes:

- The monitor's identity and authentication information, namely the monitor name, password and database.
- A list of fully qualified relations of interest. The relations are qualified as specified in section 4.3.1
- An index containing the names of the client installed primitive events (the value) for each DML operation (the key).
- An index containing the names of the monitor installed primitive events (the value) for each DML operation (the key). The set of DML operations are the complement of the client installed primitive events with respect to the INSERT, DELETE, and UPDATE operations.
- The list of modifying clients corresponding to each monitor interest relation.
- The selectivity condition for each monitor interest.

Since the interests are dynamic this module must ensure that consistency is maintained in the data stored, especially with regard to the two indices mentioned above. Whenever the owner adds or drops events in the capacity of an interactive client the monitor must take appropriate actions to ensure that all DML operations are being monitored via a primitive event. This involves intercepting interactive client create and drop trigger requests. Sections 4.4.4 and 4.4.5 on the extensions to the Persistence Manager and Drop Trigger modules respectively, describe the incorporated extensions. The chapter on implementation discusses the data structures and methods constituting the Monitor Interest module.

The result processor and queue manager modules both access data within the monitor interests. The shared nature of this data necessitates careful synchronization between the various modules accessing them in order to maintain consistency. The primary actor in modifying information is the ECA Monitor Listener which adds, deletes and modifies monitor interests. However, the persistence manager and drop trigger modules also cause changes whenever the owner of a monitored table creates or drops triggers. Such actions will change the client and monitor installed event indices.

4.3.4 Result Processor

The result processor module is a multi threaded, synchronized component of the ECA Monitor. It's function is to retrieve the relevant tuples corresponding to the current queue node in the notification queue. Notification of results to the respective monitoring clients can be a data intensive task, especially if there are several monitoring clients with large sets of monitoring interests. For this reason, a single thread is dedicated to every monitoring client. Concurrency of change notification is achieved in this way. As will be described in section 4.3.5, the queue manager identifies and signals the concerned thread of a notification to the monitor that it serves. The result processor thread uses

the information, namely the user identity, database name , table and event names to identify which monitoring clients subscribe to the changes specified for this combination. Using the user name we can identify a modifying client, the table name and database name allow us to identify the relation of interest and the version number allows us to extract the corresponding tuples from the ***R_inserted*** and ***R_deleted*** tables.

The selectivity criteria for a particular monitoring interest are applied to all tuples corresponding to the same version number are obtained from the queue manager. The result processor then sends this information back to the monitor client via the socket it shares with the ECA Monitor Listener thread servicing the same client.

4.3.5 Queue Manager

As described in the previous section each monitor client is serviced by a result processor thread that extracts the relevant updates and sends them to the client. It would be inefficient for all the result processor threads to individually poll the **NOTIFY** table to determine whether an update has taken place. Rather, whenever the *combination* of the *identity* of an interactive client and the *operation* and *relation* it acts upon exists within the set of monitor interests the **NOTIFY** table is polled once and any relevant tuples are placed in a common shared queue. As was described in section 4.2.1 the Language Filter module has been modified to glean this information from every DML operation issued by an interactive client. This queue has all the information within the **NOTIFY** table as well as a reference to all of the result processor threads whose corresponding monitor clients have specified this interest.

The queue manager runs independently of other threads and processes each inserted node. It is invoked by the *ServeOneClient* modules whenever a potential update by an interactive client is issued (described in the following sections). The queue manager notifies the respective result processor threads that wait on this object by inserting the

nodes into individual queues for each result processor. These queues are maintained by the queue manager. In this manner, selective replication of nodes within the shared, synchronized queue allows greater concurrency. Each result processor now has access to the **NOTIFY** table tuples that are of interest. As these nodes are consumed from each queue the master copy's thread references are updated. A node is removed when all the appropriate threads have processed that node.

4.4 Extensions to the existing ECA Agent

Several modules of the ECA Agent require extensions to accommodate the ECA Monitor so that they are "aware" of the monitoring process.

4.4.1 MultiClient

In order to differentiate between a monitoring and an interactive client, the initial communication or "handshake" is tailored to the type of client. Once they are differentiated the appropriate entry point is invoked: either the Monitor Listener for monitors or the *ServeOneClient* for interactive clients. Each monitor client has a result processor thread, an ECA monitor listener thread and connection information shared between these two threads.

4.4.2 ServeOneClient

This is the main entry point for any interactive client (IC) as described in section 3.3.1 and it is also the point where the **NOTIFY** table is checked for insertions indicating a primitive event firing. Since the monitor will wish to monitor **all** primitive events triggered by a modifying client a check is made with the Monitor Interest data structures if the combination of the SQL command parameters and IC correspond to any monitor interest (if it does, then it is a modifying client). In conjunction with the Language filter,

we can extract the relation and operation within the SQL command issued by the IC, and retrieve tuple from the **NOTIFY** table if necessary.

Due to the additional primitive events installed for monitoring purposes, tuples inserted by the monitor installed triggers will exist in the **NOTIFY** table. The *ServeOneClient* must be programmed to ignore these tuples by the syntax of their event names. It is for this purpose of identification that the monitor installed events and triggers have a fixed naming convention that is reserved for monitoring events. This is described in greater detail in chapter 5.

4.4.3 Language Filter

Section 3.3.2 described the language filter. In order to reduce the number of times that the monitor server checks the **NOTIFY** tables of the respective monitor interest's relations, we extend the language filter to generate the current operation/relation combination. With this information the *ServeOneClient* may determine if an interactive client's actions may be a potential candidate for observation. The monitor interest module is referenced to determine candidacy.

4.4.4 Persistence Manager

It is the persistence manager that creates events, constructs, compiles and loads the respective class files for the primitive and composite event registration. Whenever the client creates an event on a table for which a primitive event already exists, the monitor must be able to drop the installed event, allowing the interactive client to continue event creation and registration. This entails checking for a corresponding monitor installed primitive event trigger every time an interactive client creates an event. The monitor interest module allows us to quickly check this condition. The precondition, action and postcondition are:

1. **Precondition:** The monitor M1 has installed a primitive event EV_MON_OP on DML operation OP on behalf of client IC1 for monitoring relation R. This event creation is transparent to IC1.
2. **Condition**: IC1 wishes to create a primitive event IC_EVENT_OP on R for OP. The monitor interests are cross referenced to find that EV_MON_OP exists already. EV_MON_OP is dropped, the primitive event IC_EVENT_OP is created, and the monitor interest records are updates to reflect this.
3. **Postcondition**: IC_EVENT_OP exists for operation O on relation R and was created by IC1.

4.4.5 Drop Trigger

The drop trigger module is responsible for ensuring the correct dropping of primitive events and triggers as described in section 3.3.5. Whenever the client drops an event on a table which is part of a monitor interest, it is imperative that the monitor install its own primitive event on the interactive client's behalf so that updates are continuously observed and recorded in the **NOTIFY** table. The precondition, action and postcondition are:

1. **Precondition**: The interactive client IC1 has installed a primitive event IC_EVENT_OP on DML operation OP for relation R.
2. **Condition**: IC1 wishes to drop IC_EVENT_OP. Immediately after this happens the monitor creates EV_MON_OP on relation R for the DML operation OP. The monitor interests corresponding to this relation records are updated to reflect this.
3. **Postcondition**: EV_MON_OP exists for operation O on relation R and was created by IC1.

   As can be seen this is the inverse of the sequence of actions described in section 4.4.4.

4.5 Design Alternatives Considered

Other approaches that were considered in developing the architecture for a monitor were all based on the premise that we could utilize the powerful event specification capabilities of the ECA Agent. The first alternative involved constructing an event graph containing three primitive event nodes corresponding to each DML operation. The event graph would be separate from the interactive client's event graph. This entailed setting up separate directory structure for the monitors and maintaining the files for registering and raising events. The advantage of this approach is that the interactive clients actions would not affect the event graph of the monitor. Whenever the monitoring events were raised a corresponding action would execute and invoke a static method that would retrieve the results from the transient tables. However, the overhead introduced by setting up a separate event graph and maintaining the respective files outweighed its advantages.

The other approach involved setting additional events as in the adopted approach. However, the difference is that rules are also installed on the primitive events. The rule's condition is trivial and is set to TRUE as we wish the action portion to always execute. The action portion of the primitive event would invoke a static method within the monitor that extracts the relevant information from the **NOTIFY** table, applies the selectivity condition to the temporary tables and propagates the notifications and results to the appropriate monitoring clients. The downside to this approach is the excessive overhead of having to raise the event, and wait for the rule's action portion to invoke the static method. On reviewing the alternatives, it was realized that this entire process could be bypassed and the method directly invoked.

The above two approaches did not require that all DML commands sent by an interactive be processed to retrieve relation/operation information. In spite of the above requirement, the chosen alternative is more efficient as well as simple.

4.6 Summary

The design and architecture of the integrate ECA Agent/Monitor system was discussed. Each module's function and design within the purview of the ECA Monitor was described. In particular the *ECA monitor listener, the queue manager, the result processor* were the primary modules which all interacted with the shared and synchronized data managed by the Monitor Interest module. To successfully observe client updates, the ECA monitor system has to be integrated within the framework of the ECA Agent. The ECA Agent must be made monitor aware to fully support the ECA monitor. The changes to *ServeOneClient, Language Filter, Persistence Manager, Drop Trigger* and *JDBC* modules have been discussed. In the following chapter some implementation details not covered in this chapter will be elucidated. Subtle issues regarding the various cases are also discussed.

CHAPTER 5

Implementation Details

This chapter describes some of the important implementation details within the context of the modules described in chapter 4. The integrated ECA Agent and monitor have both been written in Java using Sun's Java Development Kit Version 1.3 [22]. The target database for which the framework has been implemented is Sybase.

5.1 The MonitorInterest Module

In an operational scenario, we potentially have several monitor clients each having several monitoring interests. There may or may not be overlap between the different monitor's interests. For the purposes of clarity we will refer to a monitoring client as **MCn** where n corresponds to a numerical value to differentiate between monitoring clients. Also, interactive clients will be referred to by the notation **ICn** where n is a numerical value used to differentiate between unique clients.

5.1.1 Specification

Each monitoring interest will be specified by

- A monitor name corresponding to the monitoring client (MC).
- A set of fully qualified relations which form the MC's interests. Fully qualified relations follow the syntax specified in section 4.3.1.
- For each of the specified relations a set of clients whose changes the monitor wishes to observe are specified. These clients are qualified by their usernames as identified in the Sybase database.

Parameter	Value
Monitor Name	MC1
Relation Name	Finance.Alice.EMPLOYEE
Modifying Clients	Bob, Charles
SQL predicate	Select Salary DeptID from Finance.Alice.EMPLOYEE

Table 5.1 An example specification of a monitor interest.

- A selection predicate specified as an SQL command. The selection predicate is currently limited to only the relation that is being monitored. The client also needs to specify the relation R under observation. The parameters for various parts of SQL command must be limited to the schema of the R_inserted and R_deleted tables.

As an example of the above we may have MC1 specifying a monitor interest in Finance.Alice.EMPLOYEE as shown in table 5.1:

Note that in Sybase, the login name maps onto a username within the database. Sybase allows us to easily identify the username associated with any action by calling ***user_name()***, a system defined function. Since several login names may map to a single user name the login cannot be easily identified for any action. For this reason, we restrict ourselves to identifying the user name for modifying clients.

5.1.2 Classes

These inputs must be stored in a manner that allows for easy access and modification. This is done in the *MonitorInterest* module. In its implementation we maintain two entities: the *MonitorRelations* class and the *RelationEvents* class:

MonitorRelations This class contains and manages the list of monitoring clients attached to it and the lists of their interests. The information mentioned above is encapsulated in *MonitorInterest* objects whose structure is show in figure 5.1.

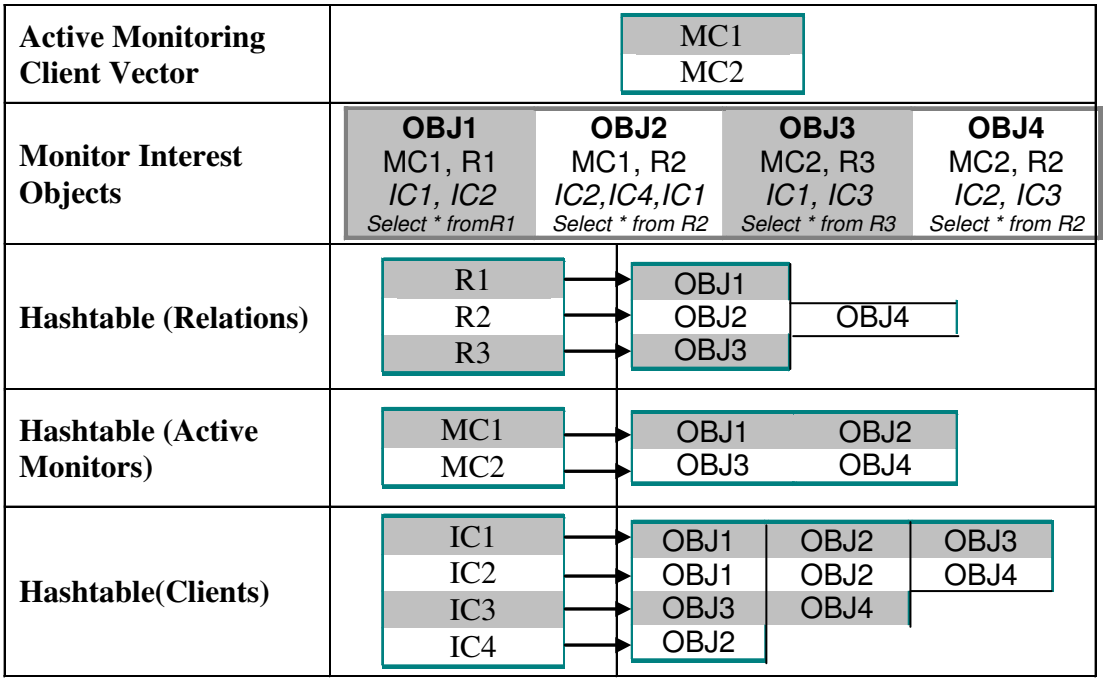

Figure 5.1 A MonitorInterest object..

These objects need to be accessed by several modules throughout the process of monitoring, including setup of monitors, initialization, result processing and notification. Since MCs may have overlapping interests the *MonitorInterest* objects also overlap with respect to some of their member elements. The *MonitorRelations* class maintains the following indices to efficiently access and update the set of *MonitorInterest* objects:

- **Fully qualified relation index**: Indexing is based on the fully qualified relation name. The list of monitors and the list of clients may be accessed with this hash table based index.
- **Active Monitors List**: This list is indexed on the monitor identity and is updated whenever a monitor client enters or exits. It also serves as a way of identifying whether there are any active monitors. This aids in quickly

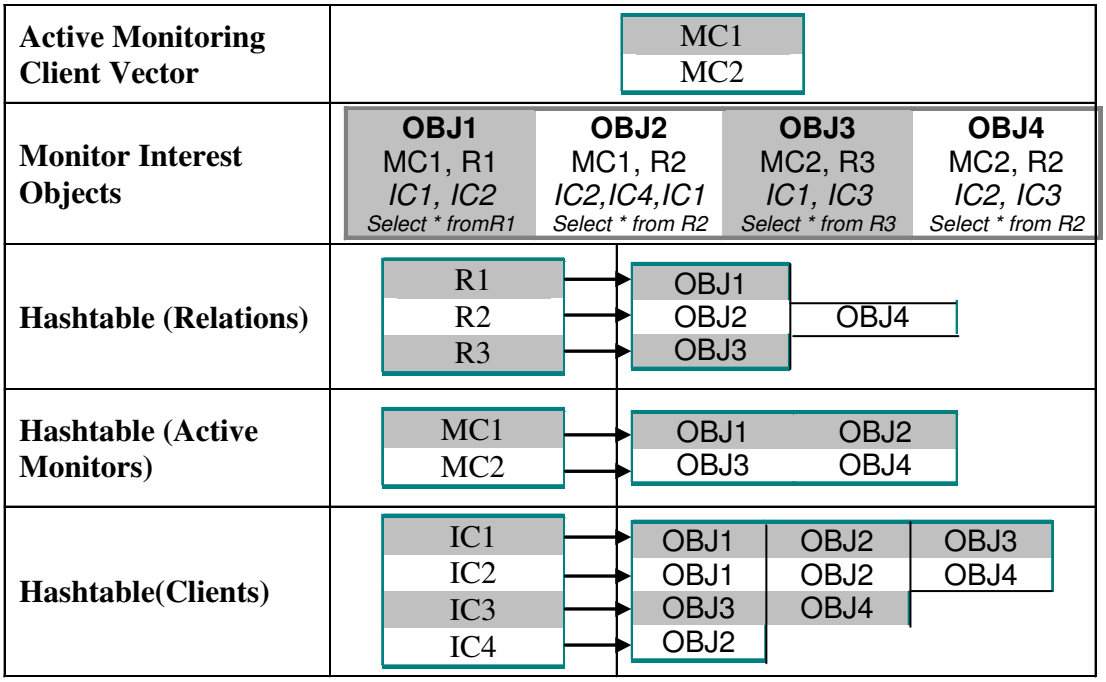

Figure 5.2 A MonitorRelation object containing several MonitorInterest objects..

identifying whether additional actions need to be taken within the persistence manager and the drop trigger modules when creating and dropping events respectively.

- **Relation lists indexed by client name**: Whenever a client makes an update through the ECA Agent client interface it is processed within the *ServeOneClient* module. To determine whether this update is a potential candidate for observation we use a hash table to determine if the given client, relation combination is part of any MC's monitor interest.

The example shown in figure 5.2 illustrates how the monitor objects are stored and referenced by these indices. These indices are updated under the following circumstances:

1. Whenever a new MC subscribes to the ECA Monitor Server with a set of monitor interests. After the monitor interests are created the index is updated by adding new client, relation list pairs and by appending new relations to existing client, relation list pairs.
2. Whenever an existing MC disconnects from the ECA Monitor Server the inverse operations take place. There is the possibility of some relation lists being empty in which case the client, relation list pair is removed from the index.
3. Some MC changes the parameters for a given monitor interest. The change may be with regard to the client list or the selection predicate. In the former case, updates to the list of clients needs to be made.

RelationEvents The other component of the *MonitorInterest* module is the *RelationEvents* class. Each relation that is a part of a monitoring interest has to have all of its DML operations observed by a trigger. Two hashtables are maintained: one for the monitor installed primtive events and the other for the relation owner installed events. The hashtable is indexed by the DML operation that corresponds to these events.

Whenever a monitor interest is added to the MonitorInterest module, the relation to be observed is checked within the SYSPRIMITIVEEVENT table. A list of primitive events that correspond to the table name and DML operation are retrieved. All of those events that correspond to the monitor's events are named with the same convention: EV_MON_DMLOperation. The corresponding trigger is named with the convention: TRIG_MON_DMLOperation. This allows easy identification by the Monitor of events that were installed by itself so as to disallow its propagation to any IC, maintaining the goal of transparency.

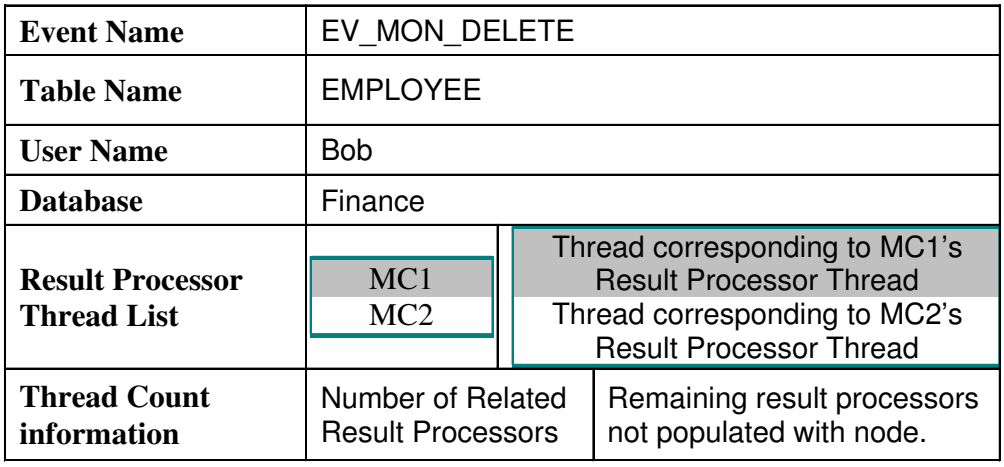

Figure 5.3 A typical queue node in the Queue Manager.

5.2 Queue Manager

As described in section 4.3.5 the queue manager has a single queue which is used to populate queues belonging to other monitors. Whenever any of the *ServeOneClients* servicing an IC diverts control to the ECA Monitor to check the respective notify tables, tuples are retrieved for all the primitive events (both IC and MC installed events) specified in *RelationEvents* for that relation. These tuples are encapsulated in a node and appended to the end of the queue. The structure of the queue node is shown in figure 5.3.

A vector of MCs with an interest in the ***"database.username.table"*** combination is also maintained. This is done with the intention of quickly identifying which individual queues the results are directed to. Once the queue manager thread gets scheduled it processes each tuple in the order that it got inserted (FIFO). For each MC in the list for a node, it appends the tuple containing only the eventname, tablename, username, database and version number. Processing of the individual queues is the responsibility of the result processor. This thread is signalled after the its respective queue has been

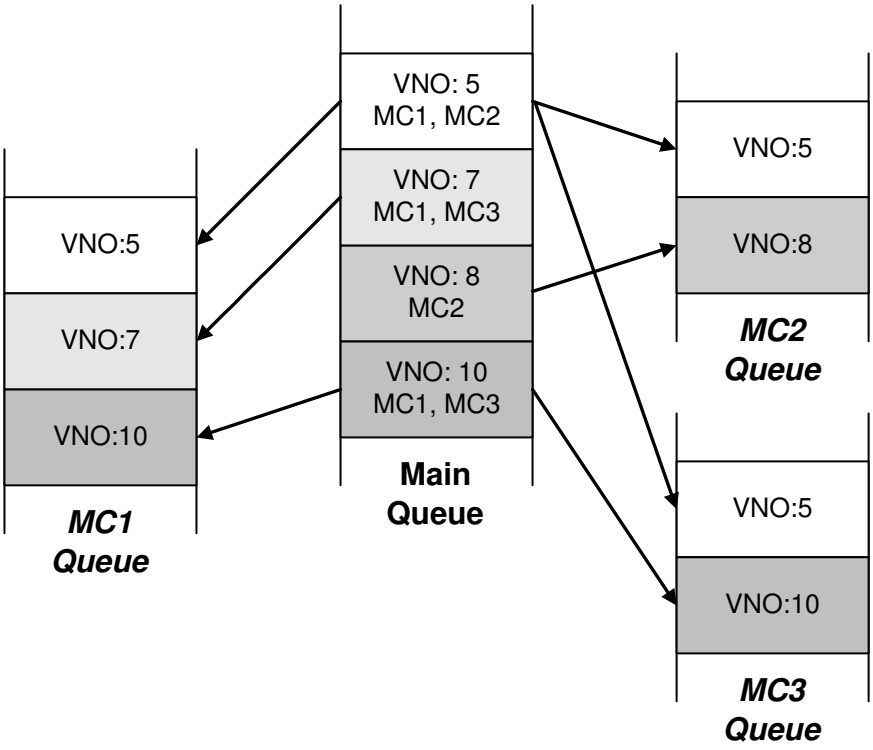

Figure 5.4 Insertions from the main queue to the individual result processor queues.

appended with a new node. Synchronization here is reasonably straightforward as this a classic producer consumer problem (the Queue Manager produces the entries which are inserted into the individual queues whereas the respective result processor threads consume them). This process is shown in figure 5.4

In section 5.1.2 we discussed how the monitor installed events were uniquely named and identified. Whenever an event occurs and its information is inserted into the **NOTIFY** table, they are retrieved, processed and deleted so that future scans of the **NOTIFY** table do not retrieve the same results. This was straightforward when only ICs (the owner) set primitive events. The ServeOneClient module would simply retrieve all tuples, raise the corresponding events and delete all of them. However, in the extended architecture, monitor installed events coexist with owner installed events.

Consider the scenario described in section 4.2.1 with the events *AliceInsertEvent, EV_MON_UPDATE* and *EV_MON_DELETE* defined on the Finance table. Whenever any client in the interest, namely *Bob* and *Charles* make updates to this table tuples are inserted into the **NOTIFY** table. This prompts a static method in the queue manager to retrieve the results into the main queue. This is achieved with a simple SQL SELECT query for all tuples corresponding to these three events. At this point, only those tuples corresponding to the monitor installed events must be deleted. Another subtlety that is involved is that during retrieval of tuples, other instances of the events may trigger and insert more tuples into the **NOTIFY** table. Since these tuples have not been retrieved, it is important to retain them for the subsequent iteration. To achieve this, for all the tuples that are retrieved, the version number, which is globally unique for every event occurrence is extracted and stored. During the deletion of processed tuples only those tuples having a version number within this extracted set are deleted. Once control returns to the ServeOneClient, there may still be monitor related events present which have to be retained. The selection is modified from a simple SELECT query to the more selective statement:

"SELECT * from NOTIFY where eventname NOT LIKE EV_MON_%".

Similarly, during deletion, only those tuples selected for processing by the ServeOneClient need to be deleted. The simple delete statement has been changed to the following statement:

"DELETE * from NOTIFY where eventname NOT LIKE EV_MON_%".

5.3 Result Processor

To understand how the result processor is able to retrieve and communicate the correct results to each monitoring client we need to understand how the occurrences of DML operations are detected and communicated to the appropriate monitors and

further how the modifying clients are identified. In the ECA Agent the **NOTIFY** table contained only the eventname, tablename and version number as described in section 3.3.4. This information was insufficient to identify the modifying client. The modified table is shown in table 5.2

EVENTNAME	TABLENAME	VNO	USERNAME	DATABASE

Table 5.2 Extended NOTIFY table captures triggering user and database

In order to support this all existing primitive events, which are converted into native Sybase trigger syntax must be changed. An example of this change is highlighted in figure 5.5. As can be seen the Sybase functions ***user_name()*** and ***db_name()*** are used to obtain the user's identity and insert it into the **NOTIFY** table. In addition we also note that the rest of the trigger syntax remains unchanged.

Whenever the monitor checks the **NOTIFY** table it inserts this information into the main queue which then places the record in the appropriate Result processor queues. The result processor thread is notified by the Queue Manager whenever a new node is appended to its queue. The result processor then iteratively processes the nodes at the head of the queue until the queue is empty before going to its wait state. When processing the information the result processor simply extracts the eventname and version number from the queue, and retrieves all the relevant results from the **R_Inserted** and **R_Deleted** table. Consider the queue node shown in figure 5.6

In this case the result processor will apply the selection condition specified as part of the monitor interest to the EMPLOYEE_Deleted table. The SQL statement may look something like this:

```
Create trigger BASICTRIGGER on PRIMEVENT for INSERT as
Begin
        update SYSPRIMITIVEEVENT set VNO=VNO+1
        where EVENTNAME=' EV_MON_INSERT'
        update VERSION set VNO=VNO+1
        insert into EMPLOYEE_inserted select inserted.ID,
                inserted.Name, inserted.DeptID, inserted.Salary,
                VERSION.VNO from inserted, VERSION
        insert into NOTIFY select EVENTNAME, TABLENAME,
                VERSION.VNO, user_name(), db_name()
                from SYSPRIMITIVEEVENT, VERSION
                where EVENTNAME='EV_MON_INSERT'
        delete from EMPLOYEE_inserted_tmp
        insert into EMPLOYEE_inserted_temp select inserted.ID,
                inserted.Name, inserted.DeptId, inserted.Salary from inserted
        delete from EMPLOYEE_deleted_temp
        insert into EMPLOYEE_deleted_temp select deleted.ID,
                deleted.Name, deleted.DeptID, deleted.Salary from deleted
        exec MON_EMPLOYEE_DELETE_PROC
End
```

Figure 5.5 Extended Trigger Syntax for monitoring triggering user and database.

EVENT NAME	EV_MON_DELETE
TABLE NAME	EMPLOYEE
USER NAME	Bob
DATABASE	Finance
VNO	5

Figure 5.6 Example Queue Node.

"Select Salary, Name from EMPLOYEE_Deleted where R_Inserted.VNO = 5"

5.4 Multiclient

In order to differentiate between IC and MC communications during setup, the *MultiClient* module, which handles server socket management, requires the initial handshake from monitoring clients to specify its nature (MC) and identity (the monitor name). The handshake is essentially a predetermined header with the monitor's name. A check is made to determine if this MC already exists and if it does a denial of service is sent as a response to the subscribing MC. If, however, it does not exist, the *Multiclient* carries out the following actions:

1. It acknowledges the IC with an acceptance response.
2. Sets up a new ECA Monitor Listener thread to handle requests from the new IC
3. Appends a record of this monitor's identity, socket information and monitor listener thread id to a list of active monitors in the system.

This information is used whenever an ECA monitor listener is to be notified (if it is in the wait state).

5.5 ECA Monitor Listener

The monitor listener processes requests from the IC, parses them and sets up monitor interests, and waits for updates from the IC with regard to its monitor interests. The requests come in the form of strings which are delimited appropriately to indicate the various monitor interest parameters. The listener behaves differently in different states. The main states that it goes into are:

Initialization: In this phase the listener create a new Result Processor thread to communicate results to the IC.

Interest Addition: A new addition is specified as a string input. The ECA listener interprets this string (embedded with header information) in order to construct the *MonitorRelation* and *RelationEvents* objects.

Interest Deletion: A deletion of an interest would involve signalling the *MonitorInterest* module of a removal of the interest. The interest is specified as a combination of the monitor's identity and the fully qualified relation. Updates in the corresponding *MonitorRelation* and *RelationEvent* are handled in the *MonitorInterest* module by invoking a deletion operation in its interface.

Interest Modification: The elements of an interest that may be updated are the client list and the selection predicate. Changes to the client list in the respective *MonitorRelation* object involve updating all the related indices. This is also handled by the *MonitorInterest* module.

Termination: Whenever an MC wishes to disconnect from the system or explicitly ends the monitoring system all the related interests must be deleted. Once again the *MonitorInterest* module is invoked to delete all the interest objects iteratively. The indices are updated to reflect the deletions and the related queue is also destroyed. The result processor thread and all associated resources are freed and the ECA monitor thread terminates. Information within the multiclient of active monitors is updated.

An important point to note is that during interest addition, deletion, and modification the shared monitor interest object being updated are locked.

5.6 Summary

In this chapter we detailed the implementation of the main modules of the ECA Monitor. The key specification supplied to the ECA Monitor Listener, the shared and synchronized data structures within the MonitorInterest modules, queue insertions and distribution by the queue manager and result retrieval by the result processor were discussed. Synchronization issues due to the multi threaded nature of the ECA monitor were highlighted. In addition, the exact changes to the existing ECA Agent modules in order to accommodate the requirements of the ECA monitor were presented.

CHAPTER 6

Conclusions and Future Work

In this chapter we summarize the major contributions of this work. We conclude this discussion by providing some avenues that we believe may provide the basis for future extensions of the monitoring system to make it a truly generalized system to suit application requirements.

6.1 Contributions

The contributions of this work can be summarized as follows:

- We identified the need for a generalized framework for the monitoring of changes to database relations by specified interactive client by monitoring clients. The requirements of transparency, timely notification of updates, scalability and generalizability to any RDBMS were set forth.
- The roles and relationship between the ***"monitoring"*** and ***"interactive"*** clients were identified within the scope of existing applications.
- Alternative approaches such as audit based approaches (Lumigent's Entegra), integrated approaches (the THOR object relational database) and traditional trigger based programming were presented and contrasted while demonstrating their inadequacy for the requirements of a monitoring system.
- Active Technology was shown to be a viable platform upon which to develop the monitoring system as a better solution to the relation monitoring problem.

- The design of the integrated ECA Agent/Monitor was presented. Each aspect of the new modules of the ECA monitor were discussed in detail and the complementary extensions to the ECA Agent were listed.

6.2 Future work

Although the current system meets the basic requirements set forth by us, we have identified several improvements and extensions that may constitute future work on this system. Some of them are detailed as follows:

- **Complex Selection predicates:** The selection predicate within the current system is limited to the relation of interest. Extensions may include the ability to support more complex SQL predicates including joins and aggregates so that more complex results and interactions may be monitored. Providing this ability would lend greater ability to tailor the retrieved results to the applications' needs.
- **Distributed Monitoring Servers:** Currently the system is limited to a single integrated ECA Agent and Monitor. The ECA Agent is able to receive updates from remote event detectors using the Global Event Detector(GED)[23].
- **Visualization:** Currently the results are simply displayed as the changed tuples of the monitored relations. Applications may wish to render these results to suit their needs. To facilitate this, a richer abstraction or API may be provided to process the results and display them. For example, time stamp information and version numbers may be used to determine which changes are "stale" and may be discarded. More current results may be suitably highlighted or emphasized to suit the application.

REFERENCES

[1] Wherify Inc., "Wherify FAQ at http://www.wherify.com," Internet Resource.

[2] U. Chakravarthy, "Rule management and evaluation: An active dbms perspective," *Special issue of ACM Sigmod Record on rule processing in databases*, vol. 18, no. 3, pp. 20–28, 1989.

[3] E. Hanson, "Rule condition testing and action execution in Ariel," in *ACM-SIGMOD Conf. on Management of Data*, 1992.

[4] E. A. Chakravarthy, U.S. and L. Maugis, "Design and implementation of active capability for an object-oriented database." University of Florida:Gainesville, Tech. Rep., 1993.

[5] R. C. Widom, Jennifer and Lindsay, "Implemented set-oriented production rules as an extension of starburst," in *Proceedings 17th International Conference on Very Large Data Bases*, Barcelona (Catalonia, Spain), 1991, pp. 275–286.

[6] S. Gatziu and K. Dittrich, "Samos: an active, object-oriented database system," *IEEE Quarterly Bulletin on Data Engineering*, vol. 15, no. 1-4, pp. 23–26, 1992.

[7] L. Li and U. Chakravarthy, "An agent-based approach to extending the native active capability of relational database systems," in *ICDE*. Australia: IEEE, 1999.

[8] Z. Song, "A generalized approach for extending the active capability of rdbmss, in database systems," Master's thesis, R&D Center, CISE Department, University of Florida:Gainesville, Gainesville, Florida, 2000.

[9] G. A. Gopalakrishnan, "Making sybase fully active: Supporting composite events and prioritized rules," Master's thesis, University of Texas at Arlington, Arlington, Texas, August 2002.

[10] Y. M. G. Aravind, "An agent based approach for extending the trigger capability of oracle," Master's thesis, University of Texas at Arlington, Arlington, Texas, May 2002.

[11] N. Subramaniam, "A mediator based approach to support eca rules in the db2 rdbms," Master's thesis, University of Texas at Arlington, Arlington, Texas, May 2002.

[12] P. H. Kim, "Notification in the thor database system," Master's thesis, Massachusetts Institute of Technology, May 2002.

[13] Lumigent Techonologies, Inc., "Lumigent Entegra Datasheet," Web Resource, 2003.

[14] Food and Drug Administration et al., *Guidance for Industry Part 11, Electronic Records; Electronic Signatures Scope and Application*, U.S. Department of Health and Human Services, FDA, August 2003.

[15] P. F. Sullivan, "Data access accountability for electronic records: Meeting 21 cfr part 11 compliance requirements for secure, computer generated audit trails," in *Data Access Accountability*, ser. Data Access Accountability. Lumigent, 2002.

[16] M. S. Mazer, "Data access accountability: Who did what to your data when?" in *Data Access Accountability*, ser. Data Access Accountability. Lumigent, 2002.

[17] J. Kadlec, "Addressing hipaa auditing requirements for data access accountability with lumigent entegra," in *Data Access Accountability*, ser. Data Access Accountability. Lumigent, 2003.

[18] P. Buneman and E. K. Clemons, "Efficiently monitoring relational databases." *ACM Transactions on Database Systems*, September 1981.

[19] S. Chakravarthy and D. Mishra, "Snoop: An expressive event specification language for active databases," *Data and Knowledge Engineering*, vol. 14, no. 10, pp. 1–26, 1994.

[20] V. Krishnaprasad, "Event detection for supporting active capability in an oodbms: Semantics, architecture, and implementation," Master's thesis, R&D Center, CISE Department, University of Florida:Gainesville, Gainesville, Florida, 1994.

[21] Server Publications Group, *Sybase Adaptive Server Enterprise Transact-SQL Users Guide*, 12nd ed., Sybase, Inc., Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608, October 1999.

[22] Sun Microsystems, "JavaTM 2 SDK, Standard Edition Documentation Version 1.3.1; http://java.sun.com/j2se/1.3/docs/index.html," Internet Resource, 2001.

[23] W. Tanpisuth, "Design and implmentation of event based subscription/notification paradigm for distributed environments," Master's thesis, University of Texas Arlington, Texas, 2001.

[24] Sun Microsystems, "Java Sockets, http://java.sun.com/docs/books/tutorial/networking/sockets/,' Internet Resource, 2001.

BIOGRAPHICAL STATEMENT

Levette Stephen Lobo was born in Mangalore, India, in 1978. He received his Bachelor's in Engineering from the University of Mysore in 1999, where he majored in Mechanical Engineering. He worked for Infosys Technologies Limited, in India, for 2 years before embarking on his graduate studies in the fall of 2001. In May 2004, he graduated from the University of Texas at Arlington with a Master of Science degree in Computer Science and Engineering. His research interests include active technology.