# THE DESIGN OF A CHANGE NOTIFICATION SERVER FOR CLIENTS OF A PASSIVE OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

by

Kathleen A. Peters

B.Sc., Simon Fraser University, 1982

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science



SIMON FRASER UNIVERSITY

July 1992

# Approval

Name: Kathleen A. Peters

Degree: Master of Science

Title of Thesis: The Design of a Change Notification Server for Clients of a Passive Object-Oriented Database Management System

Examining Committee:

Chair: Dr. Nick Cercone

Dr. Wo-Shun Luk
Senior Supervisor

Dr. Jia-Wei Han
Supervisor

Dr. Tiko Kameda
External Examiner

Date Approved: July 14, 1992

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

The Design of a Change Notification Server for Clients of a Passive Object-Oriented Database Management System.

Author: _____
(signature)

Kathleen A. Peters
(name)

July 23, 1992
(date)

## Abstract

One way to aid database clients in detecting change to shared data is to provide a notification mechanism which informs clients when the data they have an interest in has changed. Previous research has focused primarily on embedding change notification schemes in a database management system (DBMS).

This work explores the feasibility of separating the notification mechanism from the DBMS, thereby allowing an active component to be added to applications which use a passive DBMS.

This thesis presents a comprehensive design specification for an object-oriented database management system (OODBMS) change notification server. ObjectStore is used as the example underlying OODBMS.

The major contributions of this thesis are its focus on object-oriented (as opposed to relational) data change, the design of a change notification server, and the description of a language used by clients to specify conditions of interest to monitor for change. Further, we present a model of version management, and describe how our notification server can complement an application where clients are versioning data.

## Acknowledgments

I wish to express my gratitude to my supervisor, Dr. Wo-Shun Luk, for his advice and support. His patience and enthusiasm were invaluable.

I want to thank my friends and family, all of whom stuck with me through both the good times and the tough times. I especially wish to acknowledge Russ Tront, who spent hours reading various drafts of this thesis, and Alicja Pierzynska, who created graphics files for many of the diagrams. They, along with Pat Brearley, Charlotte Culver and my parents, always had words of encouragement when I needed them most.

I am also grateful to the many graduate students, faculty members, and staff of the School of Computing Science who supported me throughout my time at SFU.

## Table of Contents

## List of Tables

# List of Figures

## List of Figures cont.

# CHAPTER 1
# Introduction

Traditionally, database management systems (DBMSs) have been designed to permit multiple database (DB) clients to access and manipulate data concurrently, yet allow each individual client to behave as if they alone were using the system. This has forced the DBMS to take responsibility for data consistency, and it does this through mechanisms such as transaction management and, more than likely, a strict locking protocol. As long as transactions are of short duration (i.e., at most a very few seconds), strict locking is an acceptable method of ensuring consistency because the delay conflicting clients will experience is small.

While strict locking and a guarantee of short duration transactions are reasonable for many multi-user applications, they are not appropriate for an important, relatively new application area called "cooperative work", or "groupware". Two differences between these applications (which include co-authoring documents, software, or computer-aided designs (CAD), and multi-user data monitoring and control) and more traditional applications are:

. DB clients are collaborating, each making changes to shared data in accordance with his or her role(s) in the group. Clients know they are not working alone. They need to know what others are doing; they must be able to discover when data that affects their actions has been changed.

. DB clients need a high level of concurrency in accessing a database where long-term read and write conflicts are likely. Each collaborative application has different concurrency requirements; sometimes they can be met by relaxing strict locking (e.g.,

allowing unrepeatable reads), and sometimes by using multiple versions of data. Specialized concurrency control schemes often mean that clients must take on some responsibility for data consistency; by detecting changes made by others, clients can avoid actions which might lead, for example, to lost updates.

Cooperative applications are user-directed and event-driven systems where change occurs in a non-deterministic manner. In addition, the number of concurrent clients, their data interests, and their role(s) in the group can vary over time.

This thesis explores the question of how clients in cooperative applications can be informed of change to shared data stored in an object-oriented database management system (OODBMS). Previous research has focused primarily on embedding change notification schemes in database management systems. Although this is certainly a valid option, we feel it is important to explore alternative means of providing this functionality as it is difficult for a commercial, general-purpose OODBMS to provide for all the needs of all possible end users. The more functionality the OODBMS contains, the more difficult the system becomes to use and maintain, and the more its performance may suffer. Applications may not need to use all the capabilities embedded in an OODBMS, or may find that their requirements do not match what is provided.

The alternative is to separate the change notification functionality required by an application from the database management system the application uses. We can then build a notification mechanism with a standard "look and feel" regardless of the underlying OODBMS, and we can tailor parts of the mechanism to the needs of the application. We can add an active component to a passive OODBMS. This thesis is a design feasibility study of such an approach.

In this chapter, we first discuss in more detail the various ways change detection can be of use in cooperative applications. Then we review the two main ways database clients can detect change to persistent data. Finally, we describe the scope of this thesis, including our main objectives and system design goals.

## 1.1 WHY DB CLIENTS NEED TO DETECT CHANGE

### 1.1.1 To Watch for Special Conditions

Some database clients need to react to the occurrence of particular data patterns (e.g., alarm or exception conditions) or data manipulation operations (i.e., objects created, deleted, or modified). Clients must detect changes to discover these conditions.

### 1.1.2 To Maintain Local Copies of Data (Long-Term Reads)

DB clients access persistent data through transactions, logical units of work which involve one or more database operations on one or more data objects. The DBMS ensures that each transaction is either completely done or completely undone. Further, the DBMS ensures that concurrently executing transactions do not interfere with each other (the serializability property). Typically, the DBMS does this through the use of a locking protocol; when one transaction holds a lock on some part of the database, any other transaction requesting conflicting access to the same part of the DB must wait until the first transaction releases its lock.

Conventional DBMSs often implement a strict locking protocol: shared read and exclusive write locks, with all locks being released when a transaction commits or aborts. This provides a high level of data consistency. However, this only provides a reasonable level of concurrency if the number of clients queuing for the same data is relatively small. If the lock granularity is large (e.g., page-level locking rather than object-level locking, a

situation not uncommon in OODBMS for performance reasons [DMFV90]), concurrency is further restricted.

Data monitoring and cooperative work applications can involve DB reads of relatively long duration (e.g., user interface display of data which lasts for minutes or hours). These long-term reads delay conflicting updates until read locks are released. Concurrency can be increased if clients performing long-term reads instead make local copies of the data and then release the read locks. This action creates the potential for unrepeatable reads (i.e., local copies becoming out-of-date when other clients subsequently perform DB updates to the same data). Therefore, clients must detect change in order to keep their local copies (unlocked long-term reads) up-to-date.

### 1.1.3 To Work More Effectively with Versioning

Cooperative work applications can involve updates of long duration (e.g., hours, days, or weeks). Under a strict locking protocol, a long-term update unreasonably delays conflicting reads and other updates until its exclusive write lock(s) are released. It is also important to note that these write locks are typically transient (i.e., they exist only as long as the client process remains active and the transaction remains open). Updates taking days or weeks require persistent locks, and a way of saving "work in progress", so that clients can leave and return to a long-term update at any time. Versioning data is one way of solving all these issues.

Versioning is a mechanism (which may be embedded in an OODBMS) where a record of the changes made to data objects is kept by saving the changes as separate persistent instances, or versions, in the database. One version of a data object supersedes another rather than replaces it. If one client is in the process of creating a new version (a long-term update), other clients still have read access to the previous version(s). Concurrent updates

to the same data can be performed by creating alternate versions which may be intelligently merged at a later time.

DB clients may wish to detect when the version they have accessed is in the process of change; that is, knowing a new version is being created but not yet knowing what the changes are. Then, for example, clients can choose to avoid performing actions based on data that they know will be superseded.

Alternatively, DB clients may wish to monitor, or even participate in, the changes made to a new version.

### 1.1.4 To Support Other Methods of Concurrency Control

Versioning incurs both a storage and performance overhead, so it is desirable to restrict its use to only those applications where DB clients need to perform concurrent CONFLICTING updates, or keep a history of change.

One way to avoid versioning, yet (a) enable reads during long-term updates, and (b) provide long-term persistent locks, is to replace a long-term object update with two short-term updates: one to set a persistent application-level object lock, and one to store the changes and reset the application lock. Clients who create local copies (i.e., long-term reads) prior to the start of the long-term update will be warned that the object is undergoing change when they detect that the object's update lock has been set. Clients who read the object will know from the value of the application lock whether or not it is currently undergoing change. Clients cooperate by respecting the lock (by not doing conflicting updates) until they detect that the update has completed (when the second short-term update occurs).

The scheme outlined above not only provides a partial alternative to versioning, but allows applications to tailor long-term lock granularity to their specific needs.

## 1.2 THE TWO CHOICES: POLLING OR NOTIFICATION

Conventional DBMSs are generally passive, responding only to direct calls. Clients query the database and retrieve the information currently available. To detect changes made by others, clients must regularly poll the DBMS and compare new information with old.

Polling allows a DB client the flexibility of requesting only the specific information of interest. However, there are disadvantages:

- a certain percentage of a client's processing time is spent doing I/O and data comparison. This can be expensive if the amount of data being checked is large and/or little or no change is detected much of the time.

- if the polling interval is too long, a client may not be able to respond quickly enough to change; alternatively, too short an interval wastes the client's processing time. If the pattern of data change is unpredictable, determining the appropriate polling interval is difficult.

- polling can only provide DB clients with committed data changes. It does not allow clients to share uncommitted changes in order to, for example, merge concurrent updates on the same data.

The alternative to polling is to provide some form of notification which interrupts clients when data is changing (as yet uncommitted) and/or has changed (committed). Clients should have flexibility in specifying what changes they are interested in so that notification interrupts are kept to a minimum. [Day88] and [CH90] have shown that the notification

approach is better than polling when the amount of data being monitored is large or when timely response is required.

## 1.3 THESIS SCOPE

This thesis explores the major issues and the feasibility of implementing a change notification server for clients of passive, object-oriented database management systems.

We have chosen to focus on object-oriented DBMS for two reasons. First, we will be examining many relational DBMS change notification requirements concurrently because they are a subset of the requirements for OODBMSs. Second, OODBMSs differ significantly from relational DBMSs in that OODBMSs must support complex inheritance and aggregation hierarchies. We can make an important contribution by examining how a change notification mechanism will be affected by these additional capabilities. This thesis uses Object Design's OODBMS, ObjectStore, as an example.

This thesis presents a comprehensive change notification design specification. Prototype implementation and performance analysis are left for future work.

### 1.3.1 Objectives

This work is divided into two parts. The first part assumes that the DB schemata of potential applications do not contain versioned data, and has 3 main objectives:

- to identify and discuss object-oriented change notification requirements, including the messages which must be exchanged between clients and the notification server.

- to present the internal design of an object-oriented change notification server, and the design of relevant sections of client processes which use the notification server.

- to present a detailed description of two example applications in order to show how the change notification mechanism would be used.

The second part of this thesis presents a model for OODBMS version management, and discusses how our change notification mechanism should be extended when application DB schemata contain versioned data.

### 1.3.2 System Design Goals

This work has a number of design goals. They are:

- the condition specification, monitoring and notification mechanisms proposed should be simple but effective.

- the mechanism for specifying conditions of interest should be flexible (a specification language will be defined) and dynamic (DB clients should be able to activate and cancel condition monitoring and notification at any time).

- the condition monitoring and notification mechanism should not intervene between DB clients and the OODBMS, but rather be a completely separate server process which is informed of change and then notifies clients who have registered their interest. Thus, data exchange between the clients and the OODBMS is not slowed by an extra layer of processing, and the processing requirements of the notification mechanism are also simplified.

- the condition monitoring and notification mechanism should produce no side effects. The notification server may need to read application data, but only client processes may change the data.

- notification should occur as quickly as possible, but real-time performance is not a goal. Since cooperative work applications (as we have defined them) do not have strict real-time constraints, and monitoring/control applications exist which do not require real-time response, we have chosen not to focus on real-time performance.

- application-specific sections of the design should be identified and isolated.

## 1.4 THESIS ORGANIZATION

Chapter 2 contains a literature review, and a brief description of the ObjectStore OODBMS.

Chapters 3, 5, 6, 7 collectively present a complete description of our change notification mechanism. Chapter 3 describes what functionality is, and is not, included in the design, and why. Chapter 5 describes the messages passed between clients and the notification server. Chapter 6 describes the server's internal data components and its high-level processing algorithms. Chapter 7 describes the internal design of relevant sections of a client process which uses the notification server.

Chapter 4 presents two applications which provide examples of how our change notification mechanism can be used.

Chapter 8 contains a description of our model of version management and extends our change notification server design to applications with versioned data.

Chapter 9 contains our conclusions and suggestions for future work.

# CHAPTER 2
# Related Work and ObjectStore

In this chapter, we first present a survey of previous work on change notification and condition evaluation techniques. Our approach is compared and contrasted with the work of others. Then, we provide a brief description of the ObjectStore OODBMS.

## 2.1 CHANGE NOTIFICATION

The literature contains a number of approaches to change notification. These are discussed briefly in the sections which follow.

### 2.1.1 Notify Locks

[HZ87] describes the concept of notify locks used in ObServer (the ENCORE OODBMS object server) to enable clients to work cooperatively. A client who holds a lock on an object may ask to receive notification when other clients read, update, or have lock requests queued on the object. To make a range of data sharing and notification options possible, ObServer has five lock modes: restrictive read and write (i.e., shared read and exclusive write), nonrestrictive read and write (i.e., allowing unrepeatable reads), and a null lock (a way of receiving update notification since only lock holders can be notified).

[HZ87] does not explicitly discuss how clients are notified, or if notification is possible when objects are deleted. It is clear that clients cannot be notified when objects are inserted by others since no one but the creator has a lock on a new object, and clients cannot specify what kind of change to objects is important (they are notified of ANY change).

### 2.1.2 Active Queries

One DBMS method of data retrieval is the use of some form of passive query language (i.e., access by value). Relational DBMS queries return a set of tuples which match the query conditions; object-oriented DBMS queries return a collection of objects. To detect change, a client must regularly re-issue the passive query and compare the new result with the previous one.

An active query acts initially like a passive query - a set of tuples, or a collection of objects, is returned which match the query conditions. After that, the DBMS continuously monitors changes made to the database, checking to see if the active query result has changed, and if it has, informs the client who issued the query.

[Ris89] describes an implementation of active queries embedded in the Iris OODBMS (an extended relational DBMS). A client activates a monitor for a stored function (an Iris query) and then the DBMS checks after every committed update to see if the function result changed. If it has, the DBMS invokes the client's "tracking procedure" (a call-back) which must fetch the changes. The client avoids polling and re-evaluation of the query result (since the monitor has to recalculate the result to discover change, the DBMS stores the new result until the client retrieves it), but must still pay the penalty of I/O for the entire result and must compare new/previous data to find changes.

[SPAM91] describes an implementation of active queries embedded in the Starburst extensible RDBMS. An append-only table (essentially a journal file) is created for each active query and a fetch-wait (a blocking read) SQL primitive is used to monitor the active table waiting for new tuples to be added. A complete new result is never fetched; the client merely modifies the initial result with the contents of active table entries as they arrive.

OODBMS query languages may be expressively and/or computationally restricted [LC91] [BM91] in comparison to relational query languages like SQL. In ObjectStore, for

example, full joins are not possible (the result of a query can only be a subset of the collection being queried [LLOW91]) and methods which derive data values (e.g., sum, count) cannot be used. The conditions which could be monitored by an embedded active query language are thus restricted.

An OODBMS provides two ways to access data - by value (i.e., query language), and by navigation through object identifiers, or pointers. Thus, an active query language alone cannot provide complete coverage for specifying objects of interest by performing data access.

### 2.1.3 Active Databases

An active database monitors situations (events and conditions) and initiates some kind of response (action) when events occur and conditions are true. The response may be to:

- alert processes outside the database of data changes made by other clients.
- perform activities such as integrity checking or change propagation within the database. [ZM90] refers to this as "active data".

Triggers were an early active mechanism embedded in a DBMS. They were discussed for System R (circa 1975), and are implemented in relational DBMSs like Sybase and Ingres [Day88] [Day91]. Relational triggers are a somewhat restricted form of rules (see discussion below), and are specified separately from the table(s) they reference.

[BeM91] implements OODBMS triggers as user-defined methods encapsulated in class definitions, and as such they are always enabled for each instance of a class. [GJ91] extends the C++ class definition syntax to specify OODBMS triggers in a separate section and allows them to be activated or deactivated at any time after object creation. Only intra-object triggers are implemented by [GJ91]; that is, the condition is only evaluated when the

object associated with the trigger is changed. If the condition references other objects (an inter-object trigger), their change(s) do not cause trigger evaluation.

Recent work on active databases is aimed at embedding general-purpose imperative rules in a DBMS which specify events, conditions, and actions [Day88] [Day91] [DPG91]. Rules are seen as first-class objects; they exist separately from the objects they reference. Events may be database operations (read, insert, update, delete), temporal events, or signals from arbitrary processes. Conditions, checked when events occur, may contain database queries. If the condition part is true, the action is performed.

A rule limits notification to the OCCURRENCE of a condition. To provide the same range of notification as is possible with an active query, the following set of rules must be specified:

```
ON insert,   IF x now true THEN notify   (occurrence)
ON delete,   IF x was true THEN notify   (discontinue)
ON update,   IF x now true THEN notify   (occurrence)
ON update,   IF x was true THEN notify   (discontinue)
ON update,   IF x still true but now different THEN notify
```

Rules allow us to separate data access from condition specification for notification. Unlike active queries, rules do no force clients to obtain a copy of the previous data before change notification can be received. Rules also allow us to limit notification to particular kinds of change (e.g., only inserts and deletes).

The issue not explicitly addressed by embedded triggers or rules is that of selective activation by particular clients. If the number of active clients varies, or clients have different notification requirements, can triggers or rules be made flexible and dynamic?

### 2.1.4 Model-View-Controller (MVC) Paradigm

The Model-View-Controller paradigm is an object-oriented concept that has been used in the development of user interfaces [KP88] [Shan90] [Wiss90]. MVC divides a user interface into three types of objects:

| | |
|---|---|
| Model: | the underlying data structure(s) of the application |
| View(s): | graphical display(s) of some, or all, of the model data |
| Controller: | the interface between model, view(s), and user input |

Each view object must register its existence with its model object. A model object keeps a list of its dependent views. A typical interaction scenario for these three parts of a user interface is as follows:

> A user selects some action (via an input device like a keyboard or a mouse). The controller responds by sending a 'take action' message to the model. The model carries out the operation and then broadcasts 'I have changed' messages to all its dependent views. Each view then queries the model to get the change(s) and updates its display, if necessary.

A model object sends a message to a view object by calling a method (e.g., 'model_update()') in the view object. The method may have parameters which are used to tell the view what data in the model has changed (e.g., 'model_update(attribute_name, new_value)'), so that the view does not have to query the model to get the changes.

The MVC paradigm provides a convenient object-oriented division of labour at the concept level, but any implementation results in highly coupled model, view and controller classes. MVC is more for smaller rather than larger numbers of model objects, and more for transient rather than persistent data. If we substitute 'client' for 'view object' and 'DBMS'

for 'model object' in the above description, we can see that the resulting change notification is too coarse. If we change the substitution to 'persistent data object' for 'model object', every object in the database is then keeping track of which client is interested in it.

### 2.1.5 Selective Broadcasting

[Mey91] discusses the Unix-based Field environment [Rei90] where independent tools (clients) coordinate their actions and maintain consistency of shared data by sending and receiving messages. Clients register message patterns (conditions of interest) with a message server process; the server then filters messages from other clients so that each client receives only those messages (change notifications) of interest to it. This technique is called selective broadcasting.

### 2.1.6 Our Approach

This thesis adopts the selective broadcasting idea to design a change notification server for a passive OODBMS, specifically ObjectStore. Each client sends messages to a notification server to:

- specify what data conditions are of interest. A change specification language will be defined which combines aspects of both active queries and imperative rules. The conditions start being monitored as soon as the specification that describes them is received by the server, and remain active until either the condition specification is withdrawn by the client or the client process terminates.

- inform the server of the persistent data changes made by the client.

The persistent data change information will be selectively forwarded by the server (via notification messages) to clients who have expressed an interest in the changes.

## 2.2 CONDITION EVALUATION TECHNIQUES

Determining if an active query result has changed, or if a rule condition has become true, may be computationally expensive. It may be necessary to recompute the result after each database change if more than one relation or collection (i.e., a join) is involved.

A number of techniques have been identified for efficient condition evaluation, including:

- multiple condition optimization [Day88]. Different DB clients may specify the same condition(s) of interest, and different conditions may have common subconditions. These common parts should only be evaluated once after a DB change.

- materialization and maintenance of intermediate results [Day88]. Also known as partial view materialization [Ris89]. Conditions will be evaluated more than once, so it is more efficient to maintain partial results in memory than to regenerate them for each evaluation.

- identification of readily ignorable updates [BC79] [Day88]. It may be possible to infer that a DB change is not relevant to any condition without having to evaluate them all.

- incremental, or partial, evaluation [BC79] [Day88]. Careful analysis of conditions may reveal an ordering to their evaluation which can more quickly detect if a DB change is relevant or not. Complete evaluation should be avoided unless it is absolutely necessary.

- if a condition references an indexed attribute, [SPAM91] watches for changes to the index as a way of triggering condition evaluation.

This thesis does not involve implementation of the notification server design, but we recognize that the efficiency of condition evaluation will be a critical issue for server performance. Knowledge of these techniques has influenced the design of server data structures discussed in Chapter 6.

## 2.3 OBJECTSTORE

The following is a brief description of Object Design's OODBMS, ObjectStore. For a more detailed description, see [LLOW91].

ObjectStore's DBMS functionality is divided between two processes: the ObjectStore Server which stores and retrieves pages of data (with no knowledge of the contents of the page), and a Cache Manager which handles query and DBMS processing. There is one ObjectStore Server running on each workstation where disk storage is maintained, and there is one Cache Manager running on each workstation where client processes exist that access ObjectStore data.

The ObjectStore Server provides strict two-phase locking with a read/write lock for each PAGE. This can result in a low level of concurrency.

ObjectStore's primary access language is an extended form of C++. Extensions include new keywords (i.e., persistent, indexable), data manipulation constructs (i.e., transaction statements, iteration paths, query expressions), a collection facility (i.e., sets, lists, bags) in the form of an object class library, an inverse data members facility which maintains bidirectional object relationships, and support for versioning data.

Objects of any C++ data type may be allocated persistently (in the database) or transiently (in memory). In ObjectStore, persistence is not part of the type of an object; instead, it is selected when the object is created. For example,

```
a_train = new Train ( ... );              // transient
a_train = new( db ) Train ( ... );        // persistent
```

The ***db*** argument to the **new** operator specifies that the train object being created should be allocated in database ***db***.

Every statement that accesses data in an ObjectStore database must be within a transaction. For example,

```
database *db;
db = database::open ("/dbname");
do_transaction (0, transaction::update) {
    a_train = new (db) Train ( ... );
    } // end transaction; the commit point
db -> close ();
```

# CHAPTER 3

# Change Notification Design Requirements and Limitations

In this chapter, we discuss what functionality will, and will not, be included in the design of our change notification server. First, our basic design assumptions are outlined. Then, the requirements are given for communication between clients and the server. Of special interest is the presentation of a condition specification language used by clients to inform the server of the conditions they wish monitored for change. Lastly, we briefly discuss the issues every multi-process system must consider: coordinating process startup/shutdown, and what must be done when processes terminate unexpectedly.

## 3.1 BASIC ASSUMPTIONS

This thesis assumes that notification functionality is NOT embedded in the database (i.e., the OODBMS is passive). Therefore, to enable client change notification either (1) clients must communicate with each other directly, (2) a software layer or a server process must intervene in the data exchange between DB clients and the OODBMS in order to detect change and then notify interested clients, or (3) a server process not involved in data exchange must be told of change by clients and then notify other interested clients.

We have chosen the third option because it maintains the independence of client processes (they do not need to know about each other), data exchange between DB clients and the OODBMS is not slowed by an extra layer of processing, and the processing requirements of the notification server are simplified (no need to detect change; only need to decide which clients should be notified of change).

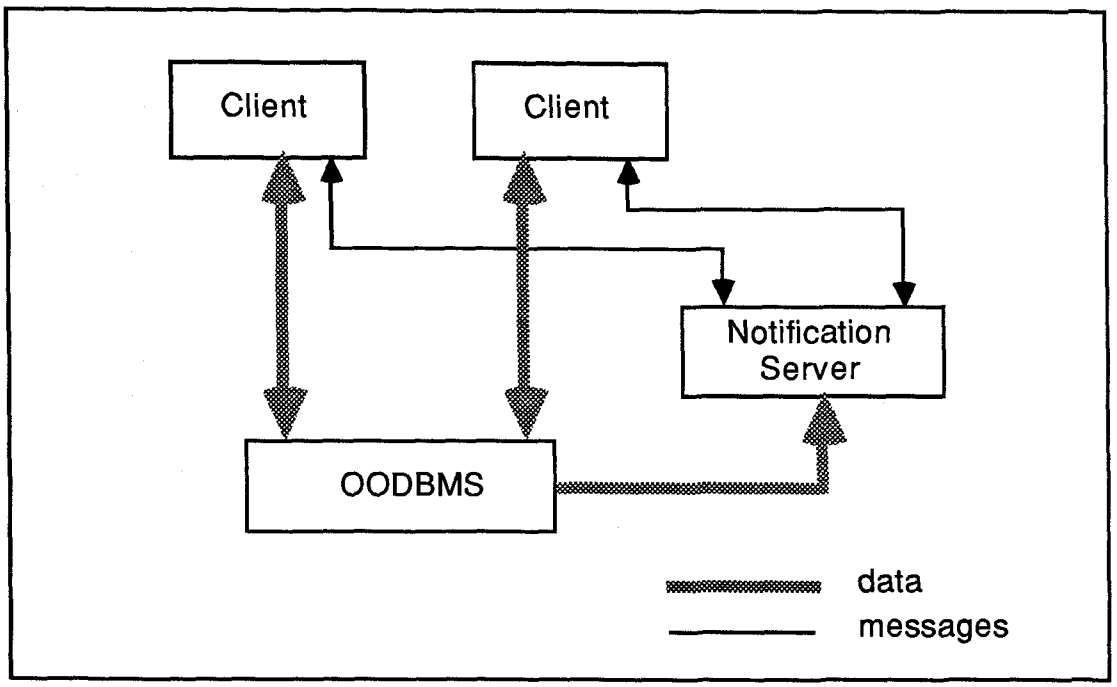



Figure 3-1 Choice of System Architecture

This thesis assumes that DB update information is only sent to the notification server once the change has been committed. This restricts notification to committed changes. Receiving uncommitted update information would add significant complexity to the design of the notification server, and since uncommitted change notification is not required for all applications, it is left for future work.

Since clients send information to the server after change is committed to the OODBMS, the server is behind (though hopefully by a small margin most of the time) in its processing of DB change. The server's view of the application data is some past state; however, its view will likely be more recent than that of the clients it needs to notify of the change. On the other hand, there is no guarantee that notification will arrive to update a client's view before the client accesses the database to, for example, update an object in its view. A client must be able to determine that its view is out-of-date; the use of a technique such as the

declaration of a sequence counter variable in objects which can be updated by multiple clients will be an important necessity.

This thesis assumes that network communication may be required. The design assumes that the OODBMS process(es), DB clients, and notification server need not run on the same workstations.

This thesis assumes the application uses one centralized database. Thus, there is no need to include a database identifier in the messages exchanged by DB clients and the notification server.

This thesis assumes that there is one notification server for each application. We realize that a centralized approach (as opposed to a distributed design) has limitations (e.g., network traffic may be higher depending on the locations of the notification server and client processes; scalability and fault tolerance are reduced). However, a single change notification server is much simpler to design, implement , and use. We have chosen not to burden this study of OODBMS change notification with distributed design overhead (e.g., more complex message passing and timing problems); we leave the study of a distributed change notification server to future work.

This thesis assumes that change notification can only be requested and occur during client process and notification server lifetimes. Long-term condition specification (e.g., to enable notification via email to users who may not be logged on when data changes) would require storage of specifications in the database; this extension to the requirements is left for future work.

## 3.2 MONITORING CHANGE IN THE DATABASE

The occurrence of particular database update events must be communicated to the notification server, as these events are key in triggering the server to check if any notification messages need to be sent.

### 3.2.1 Database Update Events

The database update events are:

. class instance creation (insert)

. class instance attribute value modification (update)

. class instance removal (delete)

During one database transaction a number of updates to different objects (i.e., correlated events) may occur. This thesis makes no attempt to monitor the net effect of multiple changes. Each object update event is assumed to be independent of any other update event. It is possible that correlated events, taken individually, cause a condition to be briefly true and then false (or vice versa). A client interested in this condition might receive two notification messages, where one or none may be preferable.

Clients send the update event messages. This means that only changes known to clients can be made known to the notification server. Change propagation (e.g., the recalculation of derived values; cascading deletes) embedded within methods in the database schema is usually considered invisible to clients, and is therefore considered invisible in this thesis.

### 3.2.2 Database Update Event Messages

One database update event message is sent by a client to the notification server for each committed object update. How much information must be included in the message?

If the message only uniquely identifies the object instance that changed and how (insert, update, delete), but not details of the change, the notification server cannot determine what

occurred by reading the information from the database. Inserted objects could be read, but versioning would be necessary to keep before/after values for object updates, and deleted objects would not exist. In addition, since the server is behind in its processing of DB change, the information it reads from the database could contain changes made after the DB update it is currently processing.

To give the server all the information it might need about a DB change, the DB update event message must contain a copy of ALL after-change attribute values for inserts and updates, and a copy of ALL before-change attribute values for deletes. In addition, the message must contain a copy of before-change values for the modified attributes of update events. If the number or size of object attributes is large, then the message length can become a communication and/or performance problem under this approach. Though it will not be feasible for all applications, this message format is nonetheless sufficient for many applications.

How will the DB update event message uniquely identify the object instance that has changed? The two possible options are:

. reference by object identifier (ID)

. reference by value, using the tuple <class name, primary key>

In theory, an OODBMS object ID is unique among all objects in the database and it never changes. In practice, some OODBMSs, including ObjectStore [LLOW91], implement an object identifier as an address, a pointer to physical or virtual storage [BM91]. But an object's address is not fixed; the illusion of an unchanged object ID is maintained by ensuring that all references to an object are updated by the OODBMS if the object's location is changed.

Thus, the DB update event messages described in this design must use object reference by value. First, the message must include the object's class name. Second, one attribute's value (or the value of several attributes in combination) must be unique among all persistent object instances of the class. The attribute may be as simple as a counter which increments each time an object is created, or it may be a more meaningful key such as employee number.

### 3.2.3 Enabling/Disabling Event Messages

It is not difficult to globally turn DB update event message passing on or off. Clients merely check the state of some flag before sending a message to the notification server. If the server is not active, or if it receives no messages, change notification is disabled. Clients then must poll to discover change.

If there are large numbers of updates but few or no conditions being monitored for change, there will be a significant overhead in message passing for which little or no benefit is being derived. It would be better if update event messages were sent to the server only for those classes directly related to conditions of interest. However, it is difficult to selectively enable or disable update event messages. If done dynamically (i.e., the server tells clients what update events to send), it would require more communication between notification server and clients, and more software complexity in clients.

A simpler solution is possible for applications where clients send a fixed set of condition specifications to the notification server - if the sending of unnecessary event messages is a significant overhead, change the client software to send only update event messages for those classes which need to be monitored. This is not an ideal solution (software changes may be necessary if a new condition specification is sent by a client) but it is easy to implement and keeps to our system design goal of "simple but effective".

## 3.3 CONDITION SPECIFICATION

Clients send one or more condition specification messages to the notification server to indicate what changes are of interest to them. Roughly, clients state "inform me if this change occurs to the result of this query".

### 3.3.1 Language Overview

We propose a flexible condition specification language. The basic format is as follows:

```
[ID#] ON <Chg_type> TO <class_name><Hchar>(<Attr_list>)
      WHERE <Predicates>
```

Figure 3-2 Basic Condition Specification Language Format

where:

**[ID#]** is a unique (for each client) number identifying the particular condition specification. It is used later when cancelling or receiving change notification for a specification.

**<Chg_type>** is any combination of the three characters 'I', 'D' and 'U'. 'I' asks for notification when an object is inserted (condition occurrence) into the query result. 'D' asks for notification when an object is removed (condition discontinued, or object deleted) from the query result. 'U' asks for notification when an object already in the query result is updated (condition remains true, but the value of some attribute(s) has changed).

**<class_name>** indicates the class of the query result (the collection of objects). The class must exist in the DB schema - a query result cannot join two or more

classes into a new class. Since the result will never actually be returned, <class_name> may be a superclass, or a parent class in an aggregation hierarchy, and then by using <Hchar> (see below) the query result can be a logical collection of more than one class.

**<Hchar>** is either null, '*', '!', or '*!'. <Hchar> provides a way of referring to a number of classes in a hierarchy with one condition specification.

If <Hchar> contains '*', then <class_name> refers to a superclass in an inheritance hierarchy, and the notification server is to monitor changes to objects of that class and all its subclasses. This convention is the same as that used by Orion OODBMS queries [BM91]. See Section 3.3.2 below for a more detailed discussion of inheritance hierarchies.

If <Hchar> contains '!', then <class_name> refers to a class which is part of an aggregation hierarchy (it contains inter-object references), and the notification server is to monitor changes to all objects in the hierarchy from the level of the class downwards. See Section 3.3.3 below for a more detailed discussion of aggregation hierarchies.

**<Attr_list>** is a selected list of <class_name>'s attributes, possibly empty, which may be given when the 'U' (update) chg_type is specified. This list tells the notification server which attributes are important to the client; without it, the server would either have to report change to any attribute, or focus only on those attributes referenced in the WHERE clause.

An attribute's type may be structured (e.g., CurDate, a structured attribute containing year, month, and day attributes). Thus, <Attr_list> may reference CurDate, in which case the client will be notified of any change to any part of the structure, or

<Attr_list> may reference a specific part of the attribute's structure (e.g., CurDate.year).

If the condition specification references an inheritance class hierarchy, only superclass attributes may be included in <Attr_list>. The attributes must be common to all the classes which will be monitored by the server.

**<Predicates>** is similar to predicates in other query languages such as SQL. The condition that <Predicates> describes must evaluate to true for any object belonging to <class_name> to be included in the query result. The following describes the general format of <Predicates>:

<Predicates> := [ NULL | <Predicate> 0{ [AND | OR] <Predicate> }n ]

<Predicate> := [ <AttrConst> <Op> <AttrConst> | <Function> ]

<AttrConst> := [<attr_id> | <constant>]

<Op> := [ < | > | = | <> | <= | >= ]

<Function> := <Function_name> (<Func_attr_list>)

The <Function> may be a general-purpose function such as 'count()' or 'sum()', or it may be an application-oriented function.

The following are some examples from a railway application:

| | |
|---|---|
| [1] ON I TO Track_Segment() WHERE status = "closed" | Notify if any Track_Segment object is added to the collection of objects whose status = "closed". |
| [2] ON ID TO Train*() WHERE cur_speed > speed_lim | Notify if any Train class or subclass object is added to or is removed from the collection of speeding trains. |
| [3] ON IUD TO Train_Schedule!() WHERE train_id = 1 | Notify if any object in the Train_Schedule aggregation hierarchy where train_id = 1 is added, or updated, or removed |

| | |
|---|---|
| [4] ON U TO Freight_Train (speed_lim) WHERE train_id = 1 | Notify if the attribute speed_lim is changed for the Freight_Train object with train _id = 1 |
| [5] ON U TO Track_Segment() WHERE in_view(West, seg_id) | Notify if any Track_Segment in the western region is changed. 'in_view' is an example of an application-oriented function |

Condition specifications differ from active queries (as defined in Chapter 2) in that they do not return an initial result (or any future result; only notification of change to the result is sent to the client) and they offer finer control in indicating what particular changes to the result are of interest.

Condition specifications differ from active database rules (as defined in Chapter 2) in that they can combine requests to monitor for condition occurrence (insert), discontinuance (delete), and object update in one statement as well as allowing notification of those changes to be requested individually if required. Condition specifications do not need to state what database update event(s) to monitor - the notification server can determine what update events cause change (if provided with some knowledge of the application DB schema). Condition specifications do not need an action clause because only one action is possible - to send a notification message to clients.

### 3.3.2 Inheritance Hierarchies

We assume that the notification server will have to handle both single and multiple inheritance in application class definitions. The following is a brief discussion of how inheritance affects the design.

Given the following inheritance hierarchy (the arrows show the superclass(s) which a subclass inherits from),

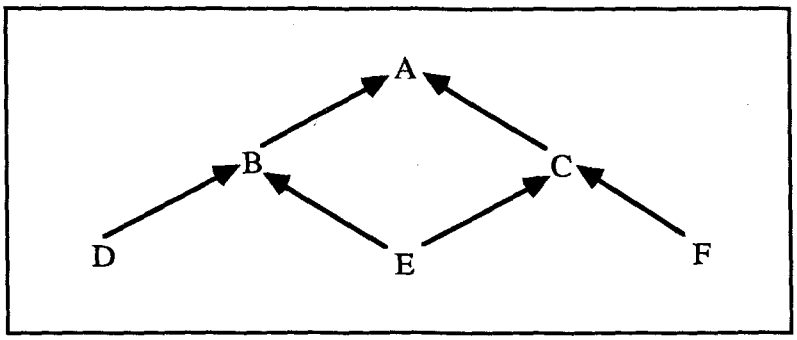



Figure 3-3 An Inheritance Hierarchy Example

if a client process issues the following two condition specifications,

[1] ON UID TO B*(<Attr_list>)

[2] ON IUD TO C*(<Attr_list>)

it should receive notification twice for change to objects of class E, except for updates where attributes in <Attr_list> are different.

On the other hand, if a client process issues one condition specification:

[3] ON UID TO A*(<Attr_list>)

it should receive only one notification for change to objects of class E. The notification server should detect common subclasses and only evaluate their changes once.

Also, the notification server can use knowledge of the DB schema to perform more efficient monitoring. If, for example, class B is an abstract base class (that is, no persistent instances of B are stored) then the server need not monitor for change to B in condition specifications [1] and [3] above.

### 3.3.3 Aggregation Hierarchies

Aggregation hierarchies occur when class definitions contain inter-object references: attributes in a class definition which may be single-object pointers or may be collections of pointers to objects. Inter-object references allow navigational data access.

For example, we could define a Car class with attributes which are single-object pointers to Engine and Body class objects:

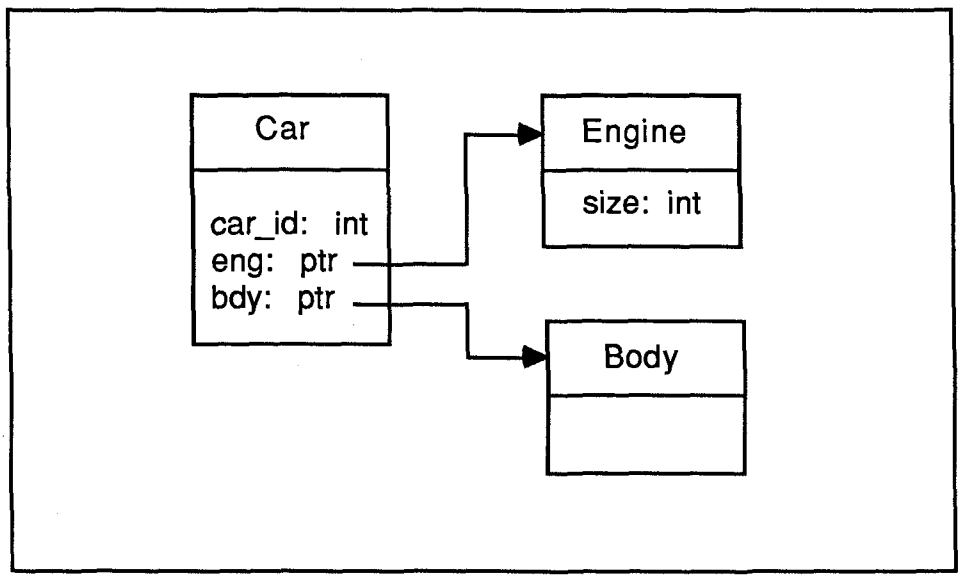



Figure 3-4 An Aggregation Hierarchy Example

In the above, the direction of the navigational access is all one-way: from Car to Engine, and from Car to Body. There is a 1:1 relationship between a Car and each of its subparts.

To be informed of any change to, for example, a particular Car hierarchy, a client would issue the following condition specification:

[1] ON IUD TO Car!() WHERE car_id = x

The condition specification's query result is the collection of all objects in a hierarchy whose ancestor node is an object of class Car with car_id = x. Inserts and deletes to the

result will refer to the named class (Car objects). Updates to the result will refer to Car attributes and, through the navigational references, to Engine and Body objects and their attributes. If, for example, the size attribute of the Engine object is changed, it is considered an update to the Car hierarchy.

Aggregation hierarchies require that the condition specification language format allow navigational references in the <Attr_list> and WHERE clause. For example, if a client wishes to be notified when a Car's Engine size is changed, the condition specification might look like:

[2] ON U TO Car!(Car.eng->size) WHERE car_id = x

On the other hand, we could focus on the Engine object itself, and use a special predicate in the WHERE clause to identify the Engine by way of the navigational access through a parent.

[3] ON U TO Engine(size) WHERE NAV=Car(x).eng

A third approach would be to define an attribute in Engine objects which points to the parent Car object (allowing two-way navigational reference). Then, the condition specification might look like:

[4] ON U TO Engine(size) WHERE parent->car_id = x

All three approaches above are valid options in the definition of our condition specification language. The same syntax is used no matter whether the reference attribute is a single pointer or a collection of pointers.

Aggregation hierarchies can be of arbitrary depth, so the notification server must be prepared to resolve lengthy navigational references.

### 3.3.4 Specifying Attribute Change for Updates

One thing missing so far from the condition specification language definition is a way of stating what kind of attribute change is of interest. Currently, the language can specify that a client is interested if, for example, the attribute cur_speed changes, but a client cannot narrow notification any further (e.g., interested only if cur_speed goes up). We can add a clause to the specification language which may be used when the 'U' (update) Chg_type is used:

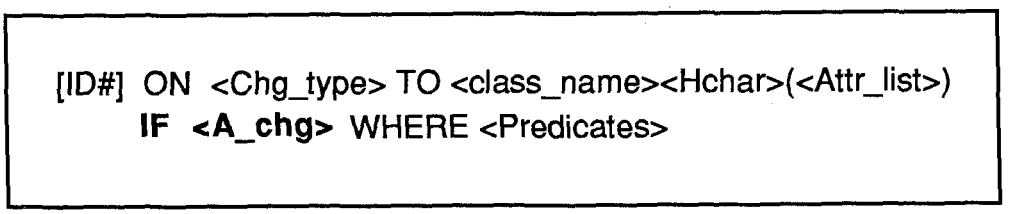



Figure 3-5 Attribute Change in the Condition Specification Language Format

where:

**<A_chg>** indicates what kind of attribute change is important. Functions to return before-change and after-change values are required. For example,

[1] ON U TO Train*(cur_speed) IF AFTER(cur_speed) > BEFORE(cur_speed)
WHERE train_id = 1

[2] ON U TO Train*(speed_limit) IF AFTER(speed_limit) > 50
WHERE train_id = 1

This fine control in condition specification may be needed for some, but not all, applications.

### 3.3.5 Placing Limits on the WHERE clause

The more general-purpose the condition specification language and the more broad its definition, the more complex the server design becomes and the more its performance may suffer. Two important limitations which any implementation of this design must consider are:

- the number of joins allowed in the WHERE clause. Joins cause a significant amount of storage and processing overhead (see the discussion in section 6.1.2.3)

- the number of predicates allowed in the WHERE clause. The more predicates that must be evaluated each time a related DB update occurs, the slower the performance.

## 3.4 ACKNOWLEDGING CONDITION SPECIFICATION

The notification server will send an asynchronous message to clients acknowledging receipt of any condition specification. Clients should always wait to receive the acknowledge message before reading any data which is intended to be updated by receipt of a notification message because:

- if the condition specification is invalid (e.g., syntax errors) it will not be monitored, and

- the notification server may have to read data from the DB to set up a monitor on multi-object conditions. The server must be monitoring the condition BEFORE the client reads any relevant data in order to guarantee that changes will not be missed. By registering interest before reading data, a client may receive notification for data not yet read, but will not end up with data which is out-of-date.

## 3.5 CANCELLING CONDITION SPECIFICATION

At any time a client may wish to cancel change notification for any or all conditions specified. The client must send the notification server a 'cancel condition' message.

## 3.6 CHANGE NOTIFICATION

### 3.6.1 Sending Notification (Server)

When the notification server detects that a particular DB change affects the result of any active condition specification, it must send a notification message to each interested client.

The client who made the DB change and sent the DB update event message will receive notification from the server for that change if the client has an active condition specification that is affected by the change. It is up to the client to weed out what it already knows from what is doesn't know when a notification message is received.

How much information must be provided to a client by a change notification message? Using the concept of selective broadcasting, the simplest way of looking at notification is that the notification server is selectively forwarding DB update event messages to clients - it could just add the affected condition specification ID# to the content of the DB update event message and send it "as is" to interested clients. However, this is not sufficient for two reasons.

Firstly, there isn't always a 1:1 correspondence between the type of change made to the DB and the type of change made to the result of a condition specification. An update (modifying the value of attribute(s)) to an object may cause an insert, update, or delete to the result of a condition specification. For example, given the following condition specification,

[1] ON IUD TO Employee() WHERE salary > 30000

if a DB change modifies an existing Employee's salary by reducing it from 31000 to 29000, the effect on [1] is to delete the Employee object from the result. Conversely, if a modification increases salary from 29000 to 31000, the effect on [1] is to insert the Employee object into the result.

Secondly, some DB changes do not involve objects of a condition specification result class, but they do cause objects to be inserted into or removed from the result. For example, the result of a condition specification with one or more joins in the WHERE clause can be affected by a change to more than one object, as in the following:

[2] ON IUD TO Employee() WHERE Employee.deptnum = Dept.deptnum
AND Dept.location = "Vancouver"

If either the Employee changes departments or the Department location changes, the result of the condition specification could change. For example, if an update to a Department object changes location from "Vancouver" to "Richmond", the effect on [2] could be the delete of one or more Employee objects from the result. In this case, not only is the cause a different type of change (an update causing deletes), but the object which changed is not of the condition specification result class, and the one DB change may cause more than one change to the result of the condition specification.

So, merely forwarding the DB update event message would not necessarily tell a client the change to the result. The notification server must be prepared to generate one, or more, notification messages which tell a client:

- exactly how the condition specification result has changed, and

. what DB change caused the change to the condition specification result. The cause is important, not only because clients may want to know, but because it will tell clients if inserts and deletes to the result are physical or logical (i.e., has an object just been created or destroyed, or is the condition just no longer true for it).

The client must not have to read any data from the DBMS to determine what has changed after notification is received because that change may already have been superseded in the OODBMS by another client update. The notification server is slightly behind in its processing of DB updates - if clients read data from the OODBMS they may not get what they expect.

### 3.6.2 Accepting Notification (Client)

Change notification must be asynchronous - client processes should not have to be blocked awaiting a change notification message. Clients must have the choice to wait for notification or do something else.

## 3.7 PROCESS STARTUP, SHUTDOWN, AND UNEXPECTED TERMINATION

A robust system must be able to handle the following situations:

. client & server startup

Under normal circumstances, the notification server should start up before any client process and shut down after all clients have terminated. If a client starts up without the server it must poll the OODBMS to detect change. It should establish connection with the server as soon as it is available though, so that its DB update event messages will reach other interested clients (since they assume that no client is making changes without notifying the server). To keep the system design

simple, this thesis assumes that clients expecting change notification will not start up if the notification server is unavailable.

. client process termination

Under normal circumstances, a client process should cancel all condition specifications it has registered with the notification server before it shuts down.

If a client process terminates unexpectedly, the notification server will detect it (when an error condition occurs while attempting to receive or send a message). The server should then cancel all condition specifications for that client.

. notification server termination

If the notification server crashes (or is shut down while in use by clients) it loses all knowledge of active clients and their condition specifications. Clients can detect a server crash when next attempting to send or receive any message as the attempt will fail. If we assume that clients always have an outstanding read waiting for messages to arrive, clients can detect a server crash almost immediately.

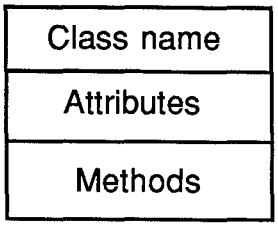
When a client detects a server crash it should start polling the DBMS. This thesis does not address how clients and a restarted server might re-establish communication.

# CHAPTER 4
# Example Applications

In this chapter, we present two different applications which help illustrate how change notification might be used. First, a brief description of the database design diagram conventions used in this, and subsequent chapters, is given. Next, the two applications, a railway network monitoring and control system, and an editor for co-authoring documents, are described.

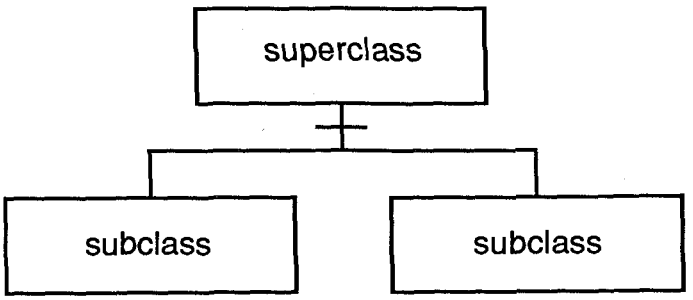
## 4.1 DATABASE DESIGN CONVENTIONS

An application's database schema design is shown diagramatically as a number of object classes, possibly connected by inheritance and/or aggregation (inter-object pointer) references. Each class definition has three parts:

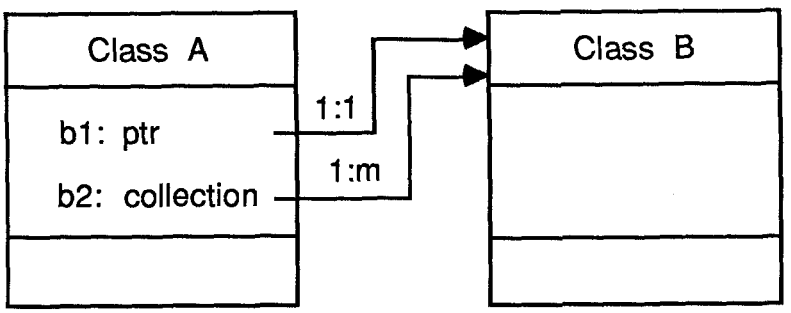



At least two methods, a 'constructor' and a 'destructor', must be present for each class. The former sets attribute values as an instance of a class is created; the latter deals with cleanup just before an instance of a class is destroyed.

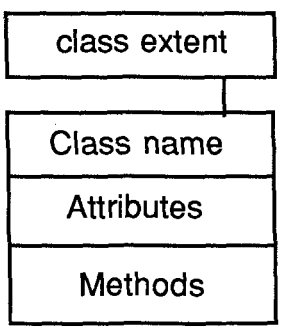
The following is an example of how an inheritance hierarchy is shown:




An example of how an aggregation hierarchy is shown is given below. An arrow indicates the direction of the inter-object reference. If the reference attribute is of a single object pointer type, the relationship is *1:1* in the direction of the arrow. If the reference attribute is of a collection type, the relationship is *1:many* in the direction of the arrow.




A class may have a persistent root collection, called an 'extent' by ObjectStore, which enables direct access to all persistent instances of the class. If a class extent is defined, it is shown in the following way:

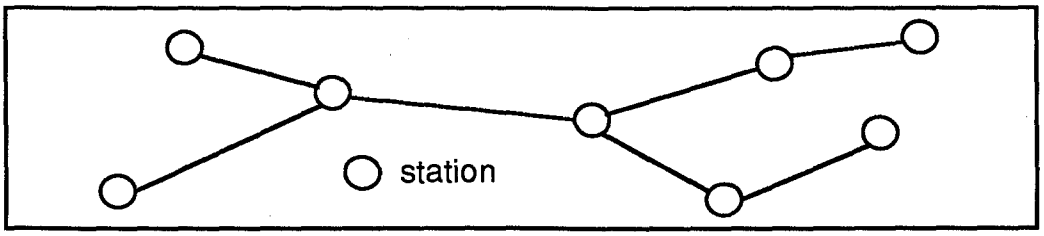


## 4.2 RAILWAY NETWORK APPLICATION

### 4.2.1 Overview

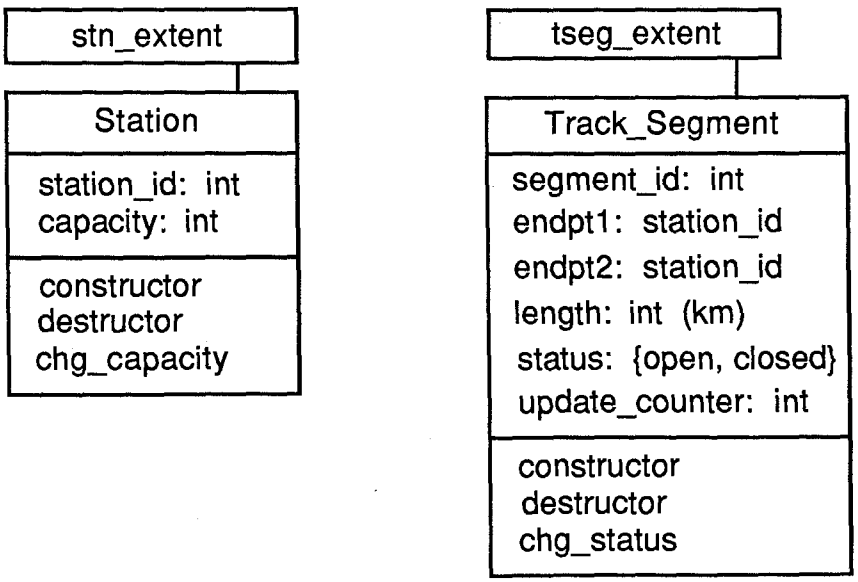
This application is the less-than-real-time operation of a simple railway monitoring and control system. Given a database containing a description of the railway network (e.g., stations, track segments connecting stations), train schedules (e.g., routes through the network), and active train information (e.g., location, speed), various DB client processes monitor and offer control options for the movement of trains in the network.

An example of a railway network is shown below. A single track connects stations. Trains can travel in either direction, but not on the same track segment at the same time.

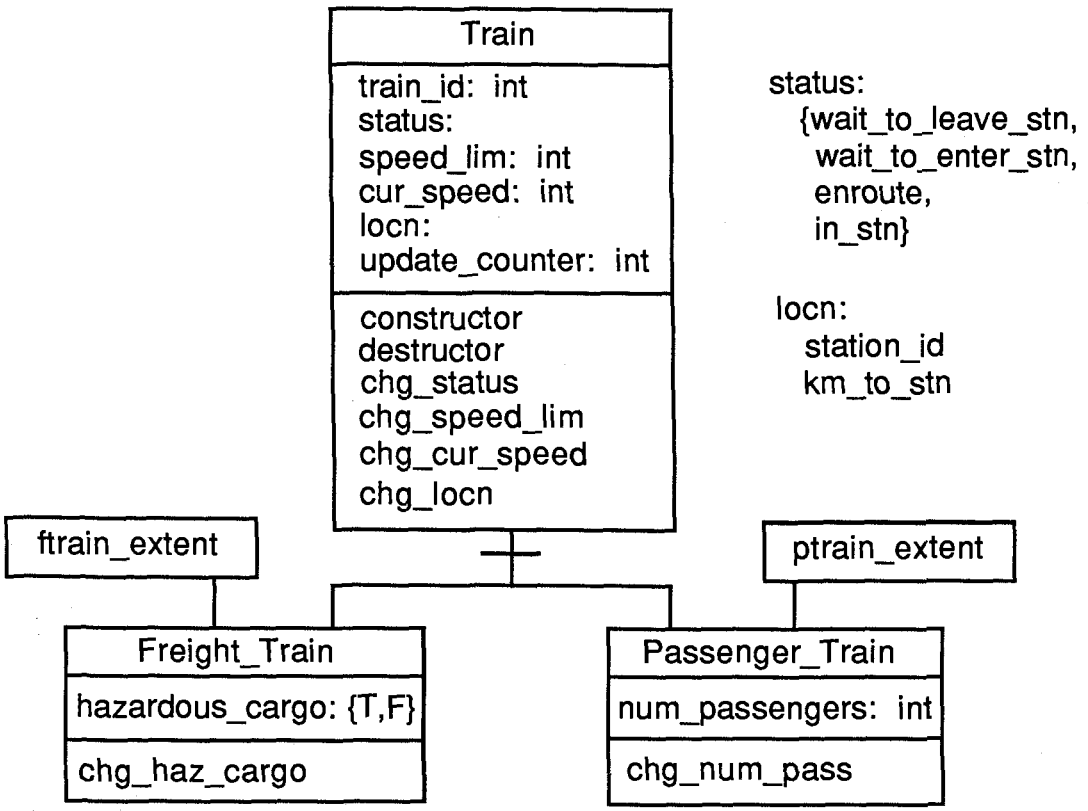



### 4.2.2 Database Design

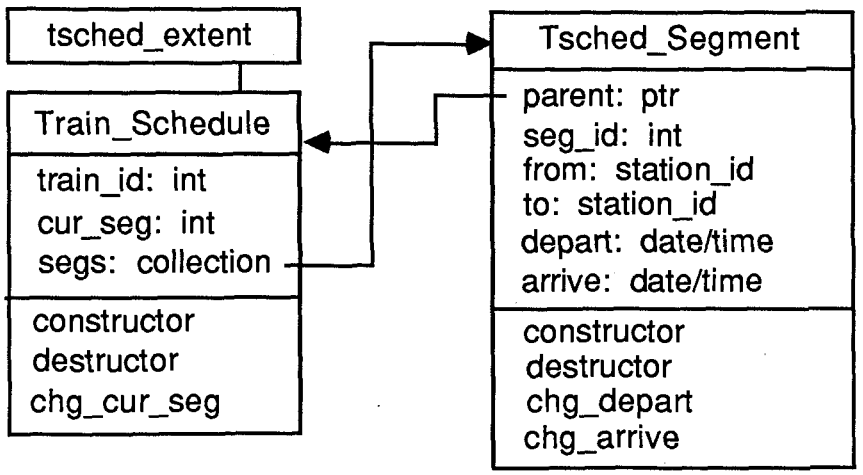
The following two classes define the railway network:



The following classes define active trains. Only instances of 'Freight_Train' or 'Passenger_Train' are stored in the database. 'Train' is simply an abstract base class.




The following 2 classes define the train schedule aggregation hierarchy:



Whenever an aggregation hierarchy is defined, the DB designer must document the insert, modify, and delete scenario for the hierarchy. This information will be important later when DB client code is written, because it can influence the way condition specifications are written, what DB update event messages are sent to the server, and what notification clients can expect.

In this case, the important points in the change scenario for the train schedule aggregation hierarchy are as follows:

. Tsched_Segment instances do not exist without a parent Train_Schedule instance. This means that if a client inserts or deletes a Tsched_Segment instance, two DB update event messages will always be sent to the notification server: one for the Tsched_Segment change, and one for the change to the parent Train_Schedule's collection attribute, segs.

. Tsched_Segment instances are not shared among Train_Schedules. This means that the navigational access path to any Tsched_Segment instance is unique. No Train_Schedule hierarchies overlap, so clients need not worry about getting more than one notification for the same change even when the server is monitoring more than one Train_Schedule hierarchy for a client.

. A Train_Schedule instance deletion automatically triggers a delete of all related Tsched_Segment instances (change propagation). This means that a client only sends the notification server one DB update event message (Train_Schedule object delete), and clients should expect only one notification message for the DB change.

. An update to the value of the attribute seg_id (the primary key) in Tsched_Segment is not allowed. Instead, two changes, a delete of one object and an insert of a new object, must be made. This means three DB update event messages will be sent to

the notification server: one for each Tsched_Segment change, and one for the changes to the parent Train_Schedule's collection attribute, segs.

### 4.2.3 Client Processes

The client processes for this application are graphic user interfaces (GUIs) for train operators and station operators.

There is one train operator GUI process for each active train. Each train operator GUI provides the following functions:

- display up-to-date train information (speed limit, current speed, current location, schedule)
- ask for, and receive, permission to enter or leave a station
- update current speed and/or location of the train

There are three station operator GUI processes: a master process which monitors the entire railway network and controls train schedules, and two region processes (west and east) which monitor subsections of the network and control the tracks and trains within their field of view.

A station GUI provides some or all of the following functions (depending on whether the GUI is a master or a regional process):

- display up-to-date network information (network diagram; train speed, location and direction of travel for each train)
- receive requests, and grant permission, for trains to enter or leave a station
- update train schedule

. update train speed limit

. update track segment status (open, closed)

### 4.2.4 The Use of Change Notification

The pattern of data access and manipulation for clients of this kind of application is as follows:

. long-term reads for GUI display, and

. short-term updates occurring when track and train information is updated

#### 4.2.4.1 Maintaining Local Copies

Client processes avoid long-term read locks by making local copies of data from short-term reads. Change notification is used to keep the local data copies up-to-date.

A train operator GUI process (in the following examples, a Freight_Train) sends the following condition specifications to the notification server in order to keep its local data copies up-to-date:

[1] ON U TO Freight_Train(speed_lim) WHERE train_id = x

[2] ON U TO Train_Schedule!() WHERE train_id = x

A station operator GUI process (in the following examples, for the western region) sends the following condition specifications to the notification server in order to keep its local data copies up-to-date:

[1] ON IUD TO Train*(cur_speed,locn) WHERE in_view(West,locn)

[2] ON U TO Track_Segment(status) WHERE in_view(West,seg_id)

Both examples above use a convenient application-oriented function, 'in-view', to limit the object instances included in the result to those within a particular station operator's network field of view.

#### 4.2.4.2 Detecting Special Conditions

Change notification is also used to detect the occurrence of conditions which require special attention.

A station operator receives a train request to enter or leave a station by receiving notification that a train is waiting:

[3] ON I TO Train*() WHERE status = wait_to_leave_stn OR status = wait_to_enter_stn

A train operator receives permission to enter or leave a station by receiving notification that a station operator has changed the train's status to enroute (leave) or in_stn (enter):

[3] ON I TO Freight_Train() WHERE train_id = x AND

(status = enroute OR status = in_stn)

A station operator can also monitor alarm conditions like the following for detecting speeding trains:

[4] ON I TO Train*() WHERE cur_speed > speed_limit

### 4.2.5 Avoiding Lost Updates

Train or station operators can choose to update the information they see displayed on the screen. The display reflects the contents of the GUI's local data copies. Theoretically, these copies are kept up-to-date by change notification, but it is possible to request an update to an object which is out-of-date (i.e., notification has not yet arrived).

To detect out-of-date data, an update_counter variable is maintained for each object which may be updated by more than one client. At the start of an update transaction (i.e., after locking the object), DB clients should verify the state of the object by checking that the value of the update_counter in the DB and in the local copy match. If they do not match, the client must update its local copy before starting the update (and then be prepared for a notification message that contains redundant information).

## 4.3 DOCUMENT CO-AUTHORING APPLICATION

### 4.3.1 Overview

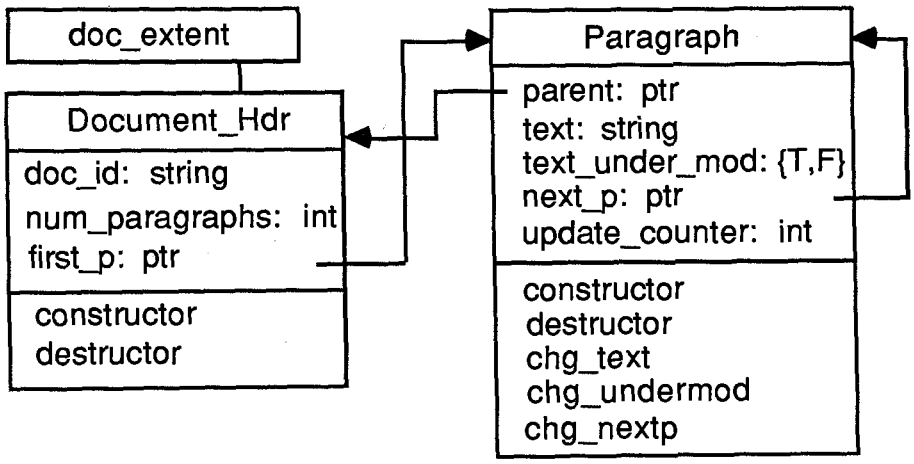
This application is a very simple editor that allows more than one user to concurrently update a document.

A 'document' consists of a document header, and any number of paragraphs (text only) linked together in a particular order. Users can insert, update, delete, and re-arrange paragraphs.

Users cannot update the text of the same paragraph concurrently. They will be informed when another user has exclusive access to a paragraph and must cooperate by waiting until the changes have been committed.

### 4.3.2 Database Design

The following two classes define the document aggregation hierarchy:




The important points in the change scenario for the document aggregation hierarchy are as follows (the significance of these points was discussed previously in Section 4.2.2):

- Paragraph instances do not exist without a parent Document_Hdr instance
- Paragraph instances are not shared among Documents
- A Document_Hdr instance delete automatically triggers a delete of all related Paragraph instances (change propagation)

### 4.3.3 Client Processes

The client processes for this application are graphic user interfaces (GUIs) that allow users to display and edit documents.

### 4.3.4 The Use of Change Notification

The pattern of data access and manipulation for clients of this application is:

- long-term reads for document display
- short-term updates for paragraph insert, delete, and re-ordering

. long-term updates for changes to existing paragraph text

#### 4.3.4.1 Maintaining Local Copies

A client process sends the following condition specification to the notification server in order to keep its local data copies up-to-date:

[1] ON IUD TO Document_Hdr!() WHERE doc_id = x

The client process will be then informed of any change to the document. Note that by issuing a condition specification that monitors the entire document, clients will receive notification of their own changes as well as those of others.

#### 4.3.4.2 Long-term Updates Without Versioning

When a user is updating the text of an existing paragraph (generally, a long-term update), the DB object cannot be write-locked as this would delay concurrent reads. To allow concurrent reads and inform other users that the text is changing, the client process wanting to change a paragraph first performs a short-term update to set the specific paragraph's 'text_under_mod' variable to true. The notification server is informed by the client who performed the change that a committed update to 'text_under_mod' took place. This information is passed on to interested clients (those who have issued the condition specification [1] above).

When the updated text is committed some time later, interested clients are informed (again via [1] above). If the 'text_under_mod' variable is now false, then another client may begin a long-term text update; otherwise, the first client still has exclusive update access, but has chosen to let others see an intermediate result.

A non-conflicting concurrent update is allowed when a client is updating the text of a paragraph and another client wishes to change the 'next_p' variable of that paragraph (inserting a new paragraph, or rearranging paragraph order). The change to 'next_p' is a short-term update. The update_ counter variable is used by the client process doing the long-term update to determine that a change has occurred to 'next_p' if notification does not reach it before the text update is committed.

There are two interesting things to note in the above:

- there is no way for users to queue for access to an object for long-term update. When 'text_under_mod' is reset, ALL interested users are informed and whoever then sets 'text_under_mod' first gets to perform the next long-term update. This is acceptable for the application, and is no worse than what is possible if ObjectStore versioning is used since queuing on long-term locks isn't possible with versioning either.

- what happens if a client process terminates, leaving one or more 'text_under_mod' variables set to true? For this application, it would be convenient if the application locks did not remain set beyond the lifetime of the process that set them. If a client process terminates normally it will reset its application locks (in the spirit of cooperation); if it terminates abnormally, the application lock(s) it leaves behind prevent other clients from updating the data. This again is no worse than what is possible if ObjectStore versioning is used.

  The notification server (if it does not also terminate) will detect the client process' termination, and could reset any outstanding application locks, but then a fundamental design goal (that the notification server does not modify the application data) is violated. The server is a selective broadcaster, not a lock manager.

An approach that does not violate design goals would be to have the notification server inform other clients that a client terminated leaving application lock(s) set. The remaining clients could then release or override the lock(s) if they wished to. This approach would broaden the definition of "change" that the server is monitoring, and is left for future work.

# CHAPTER 5

# Design of the Interface Between Clients and the Notification Server

In this chapter, we first describe the general architecture of applications which use ObjectStore and our notification server. Then, we present the detailed design of client/server communication. The content of each message sent between DB client processes and the notification server is described.

## 5.1 OVERVIEW

DB client, Objectstore, and notification server processes are expected to run on SUN workstations using the UNIX operating system. These processes may reside on the same or separate workstations; our server design allows for distributed applications.

The interprocess communication (IPC) mechanism in UNIX [Hor86] [CS92] is based on socket pairs - reference points to which messages may be sent and from which messages may be received. The notification server design requires sequenced, reliable interprocess communication, so the stream socket protocol is used.

One example of an application architecture is shown in Figure 5-1: two workstations, one where two DB client processes are run, and one where the ObjectStore server and the notification server are run. The ObjectStore server process is merely a disk page manager. An ObjectStore Cache Manager process (one on each workstation where other processes exist that access ObjectStore data) has knowledge of the DB application class schema, and

therefore it maps ObjectStore data into a client's virtual memory and does query processing.

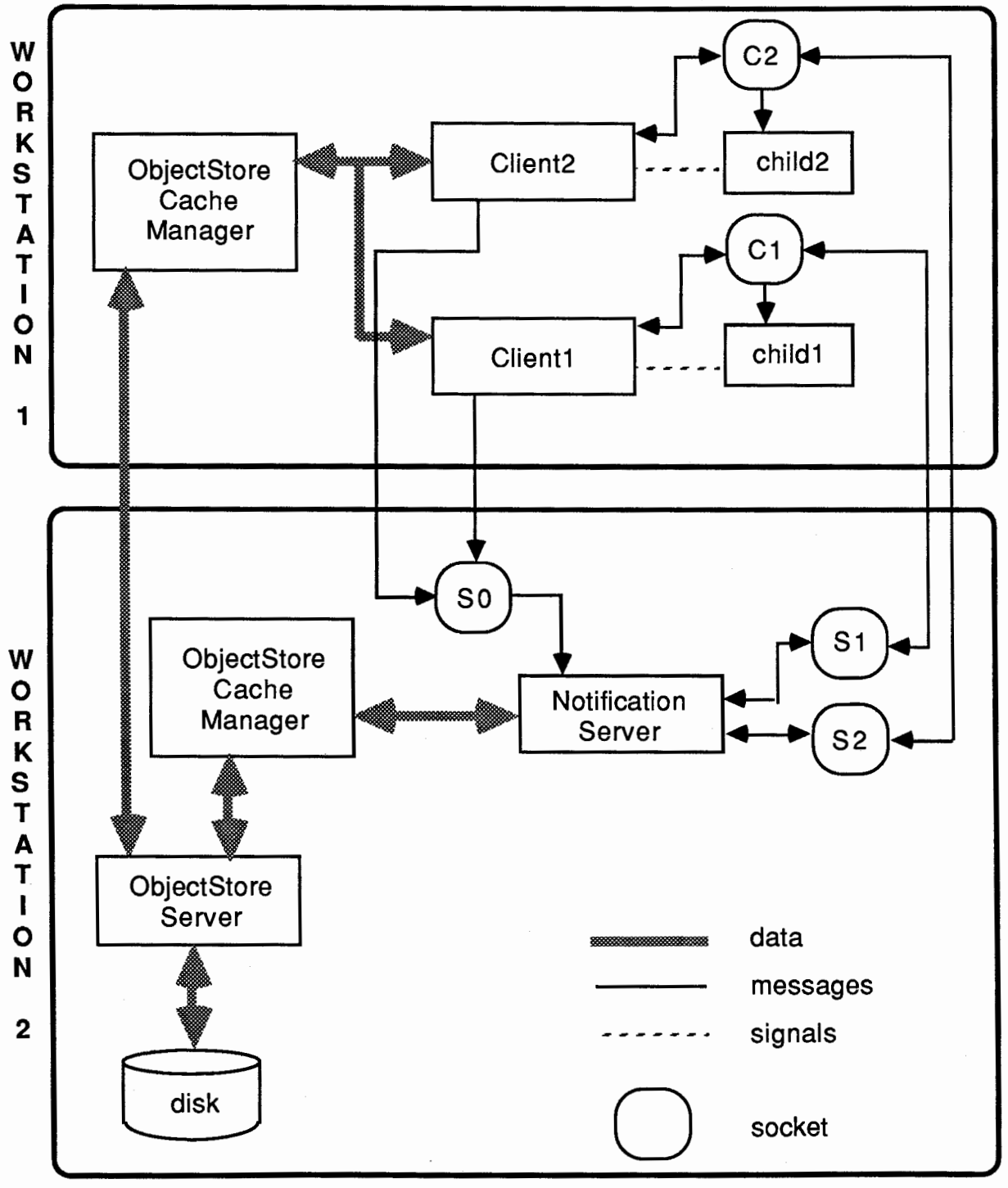



Figure 5-1 Application Architecture Example

Each client establishes communication with the notification server by issuing a connect call to socket S0. When the server accepts the connection, a new server socket (Sn) is created and paired with the client's socket (Cn) so that the server may continue listening for other connection requests through the original socket (S0).

Figure 5-1 shows one way client processes can be structured to receive asynchronous messages from the notification server. Each client process forks a child process to listen at the client socket (Cn) for messages from the server. The child is blocked waiting for a message. When a message is received, the child interrupts (signals) the parent process, and then the parent reads and processes the message from the server. Alternately, if client processes use a window management system, a different architecture is possible (see Chapter 7).

Clients send messages to the notification server through their sockets (C1..Cn). The notification server scans its half of the client socket pairs (S1..Sn) waiting for messages to arrive. Messages must be processed oldest-first, using timestamps, so that the server stays in sync with (though slightly behind in real time) the order in which condition specifications were issued and database updates were made.

The use of timestamps within a distributed system has a drawback - each workstation could have a (perhaps largely) different system time in which case the server's ordering of messages might not accurately reflect the sequence of events. This thesis assumes that workstation system times are closely synchronized and/or that the applications do not generate closely spaced DB updates where ordering is critical (that is, if ordering is occasionally out of sync nothing vital is lost).

## 5.2 CLIENT/SERVER COMMUNICATION

All message passing between DB client processes and the notification server is done asynchronously.

| From Clients to the Server | From the Server to Clients |
|---|---|
| DB Update Event Messages | Change Notification Messages |
| Condition Specification Messages | Acknowledge Condition Specification Messages |
| Cancel Condition Specification Messages | |

Table 5-1 Messages Passed Between Clients and the Server

Table 5-1 above summarizes the interface between DB clients and the notification server. The following sections discuss the format and content of the messages.

### 5.2.1 Database Update Event Messages

Clients send one DB update event message to the notification server for each committed change (insert, update, delete) to an object.

The DB event message format is given in Table 5-2 below. The first six fields are part of every message. The remaining fields are only present if the message describes an object modification event (an 'update', rather than an 'insert' or 'delete').

For example, if a client changed a railway track segment object's status variable value from "open" to "closed", the DB update event message would look something like the following:

U<timestamp><msglen>U,Track_Segment,1,1,3,100,closed,6,status,open

after image — before info

| FIELD | DESCRIPTION |
|---|---|
| Message type<br>Timestamp<br>Message length<br>'I' \|'D' \| 'U'<br>Class name | 'U' (DB update event message code<br>commit time<br># bytes<br>object insert, delete, or update flag<br>object's class |
| After-change attribute values | value of each object attribute<br>(ordered as listed in the class definition) |
| Attribute name *<br><br>Before value * | if change flag = 'U', full name of the<br>attribute that changed<br>if change flag = 'U', value of the<br>attribute before change |

* field repeated for each attribute modified

Table 5-2 DB Update Event Message Format

There are several things to note about the DB update event message format:

. after-change attribute values must be listed in some known order so that attribute names do not also have to be specified. We have chosen to list the attributes in the order in which they are declared in the class definition. If the class is part of an inheritance hierarchy, attributes are listed in the order of inheritance (i.e., superclass attributes before subclass attributes, taking care to note where superclass attributes are overridden by subclass definitions).

. where attributes have structured types, the before-change attribute names must be full names (e.g., CurDate.year) to prevent ambiguous attribute identification.

. attributes values can have a long length (e.g., a 512 x 512 image, or a 1024 byte text string). Including both the before-change and after-change values for these kinds of attributes can make the DB update event message length uncomfortably long (maybe too long for IPC message buffering or adequate performance). The after-change value must be sent, but it may not be necessary to send the before-change value for these long-length attributes.

The before-change value is sent so that the server can appropriately evaluate the WHERE clause of a condition specification (to see if the object was/wasn't in the result before the change, and to see if the object is/isn't in the result after the change). How likely is it that a condition specification's result will be restricted or selected by a WHERE clause which uses a long-length attribute? It depends on the application, but when an application's clients will not use a long-length attribute in any condition specification's WHERE clause, clients need not include the before-change value in the DB update event message.

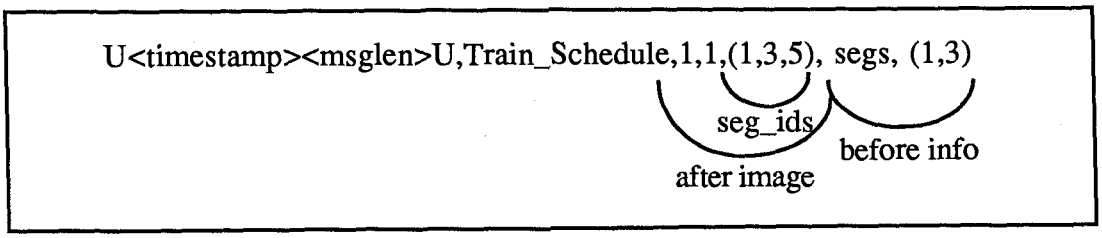
In any case, if a long-length attribute is used, as in the following condition specification

[1] ON IUD TO Paragraph() WHERE contains(text,"Vancouver")

clients must be prepared for slower server performance since the evaluation of such a predicate will be costly.

- some attributes may be single-object pointers or collections of pointers. To put the actual value of such attributes (zero, one, or a number of storage addresses) in the DB update event message is of no use, since the addresses are only valid within a client's DB transaction. Instead, the object(s) the pointer(s) refer to are identified by value (i.e., primary key).

  For example, if a client adds a new Tsched_Segment object (seg_id = 5) to a Train_Schedule aggregation hierarchy, the DB update event message would look something like the following:




### 5.2.2 Condition Specification Messages

Clients send condition specification messages to the notification server in order to indicate what changes are of interest. The condition specification message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'S' (condition spec message code) |
| Timestamp | time message sent |
| Message length | # bytes |
| Condition specification | format as discussed in Section 3.2 |

Table 5-3 Condition Specification Message Format

### 5.2.3 Acknowledge Condition Specification Messages

The notification server sends a message to clients acknowledging receipt of each condition specification. Clients should always check that a 'valid' status code is returned. The acknowledge message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'A' (acknowledge message code) |
| Timestamp | time message sent |
| Message length | # bytes |
| Condition ID | client's unique condition spec. # |
| Status code | 0 (valid and now active); 1 (error) |

Table 5-4 Acknowledge Condition Spec. Message Format

### 5.2.4 Cancel Condition Specification Messages

At any time a client may wish to cancel change notification for any or all conditions specified. The cancel condition message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'C' (cancel message code) |
| Timestamp | time message sent |
| Message length | # bytes |
| Cancel flag | '1' or 'A' (all) |
| Condition ID | the # of the condition spec. to be cancelled (if cancel flag = '1') |

Table 5-5 Cancel Condition Specification Message Format

### 5.2.5 Change Notification Messages

The notification server sends notification messages to clients, one for each change to the result of an active condition specification. The notification message format is given below. The comments about attribute names and values made in section 5.2.1 apply here as well.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'N' (notification message code) |
| Timestamp | time message sent |
| Message length | # bytes |
| Condition ID | client's unique condition spec. # |
| 'I' \| 'D' \| 'U' | result change (insert, delete, update) |
| CAUSE: | |
| DB update event | 'I' \| 'D' \| 'U' |
| Class name | class of the object that changed |
| Primary key | unique ID of the object that changed |
| Class name | result object's class |
| After-change attribute values | value of each result object attribute (ordered as listed in the class definition |
| Attribute name * | if cause event = 'U', full name of the attribute that changed |
| Before value * | if cause event = 'U', value of the attribute that changed |

* field repeated for each attribute modified

Table 5-6 Change Notification Message Format

# CHAPTER 6
# Internal Design of the Notification Server

In this chapter, we first present a description of the notification server's major data components by discussing their content and providing examples of their use. Then we present the server's major processing algorithms.

It is not our intention here to completely specify the server's detailed design (ready to code). Instead, we only wish to show that it is feasible for the server to accomplish what we've said it will do, and show the directions an implementation of our approach would take.

## 6.1 THE FOUR MAJOR DATA COMPONENTS

We have identified four major data components of the notification server as shown in Figure 6-1 below.

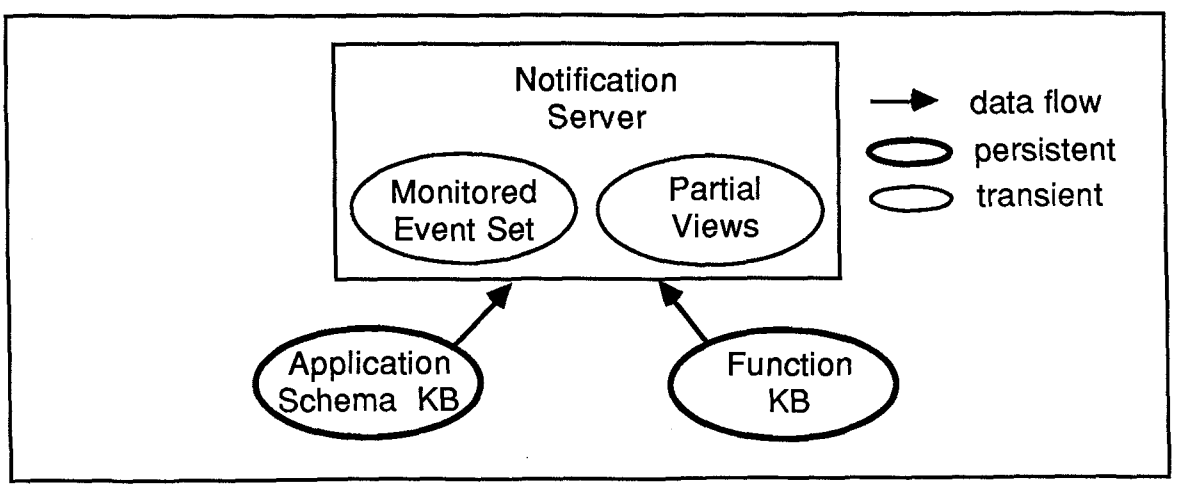



Figure 6-1 The Server's Four Major Data Components

The application schema knowledge base (KB) provides descriptive information about the application's database schema. Condition specifications are mapped into the monitored event set. Partial views are collections of application data objects which must sometimes be kept by the server when monitoring complex condition specifications. The function knowledge base provides descriptive information about functions (e.g., count(), in_view(), etc.) which can be used in the WHERE clause of condition specifications.

These data components are described in the sections which follow, where the high-level content of the data structures is defined. See Appendix B for a more detailed description of the application schema KB and the monitored event set.

### 6.1.1 The Application Schema Knowledge Base

Descriptive information about the application's DB schema is required by the notification server so that it can validate condition specifications and can determine the appropriate DB update events to monitor.

Relational DBMSs maintain a 'system catalogue' which contains information about every table, every table attribute, and every table index in the database [Date90]. The system catalogue's tables can be queried just like application tables; thus, DB clients are able to retrieve information about the structure of the application DB schema.

The ObjectStore OODBMS (release 1.2) stores application schema information in the database, but its structure is not documented and its content is not accessible by application client processes. Therefore, we have had to define our own application schema knowledge base (KB) structure. Its content must be generated for each application system that intends to use the notification server. If the ObjectStore schema information becomes accessible in a future release, its structure should be compared with our KB; if ObjectStore provides the

necessary information, the notification server design could be modified to use the ObjectStore data (but that would make it more ObjectStore-dependent).

Figure 6-2 below shows the classes that form the high-level application schema knowledge base structure.

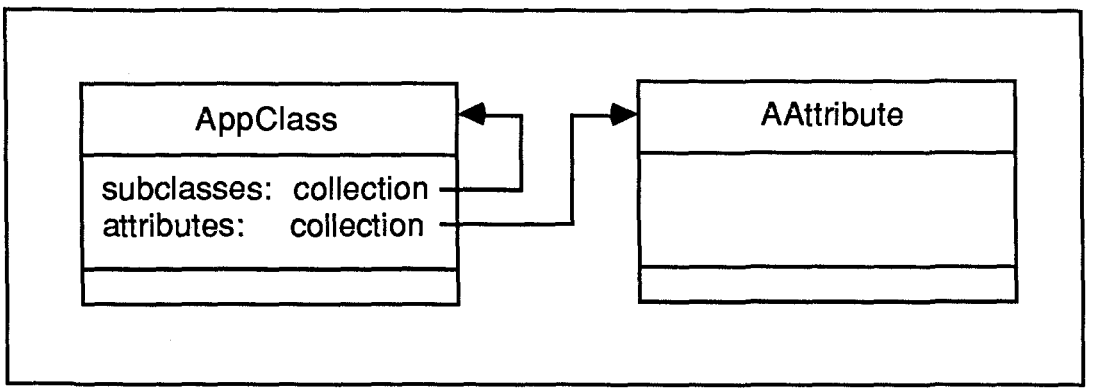



Figure 6-2 High-Level Application Schema KB Structure

There is one AppClass instance for each application schema class. The subclasses collection documents class hierarchy connections. The attribute information in AAttribute instances is needed for validation, and to document aggregation hierarchies (i.e., attributes may be pointers, or collections of pointers, to other objects).

### 6.1.2 The Monitored Event Set

Condition specifications are mapped into a dynamic data structure called the monitored event set. The notification server compares incoming DB update event messages with the contents of the monitored event set, looking for matches. When a match is found, one or more clients will then be sent a change notification message.

### 6.1.2.1 Data Structure Overview

Figure 6-3 shows the classes that form the high-level monitored event set data structure. Generally, the structure maps each active condition specification into one or more DB update events, each of which has a condition to be evaluated when the event occurs and action(s) to be taken when the condition is true.

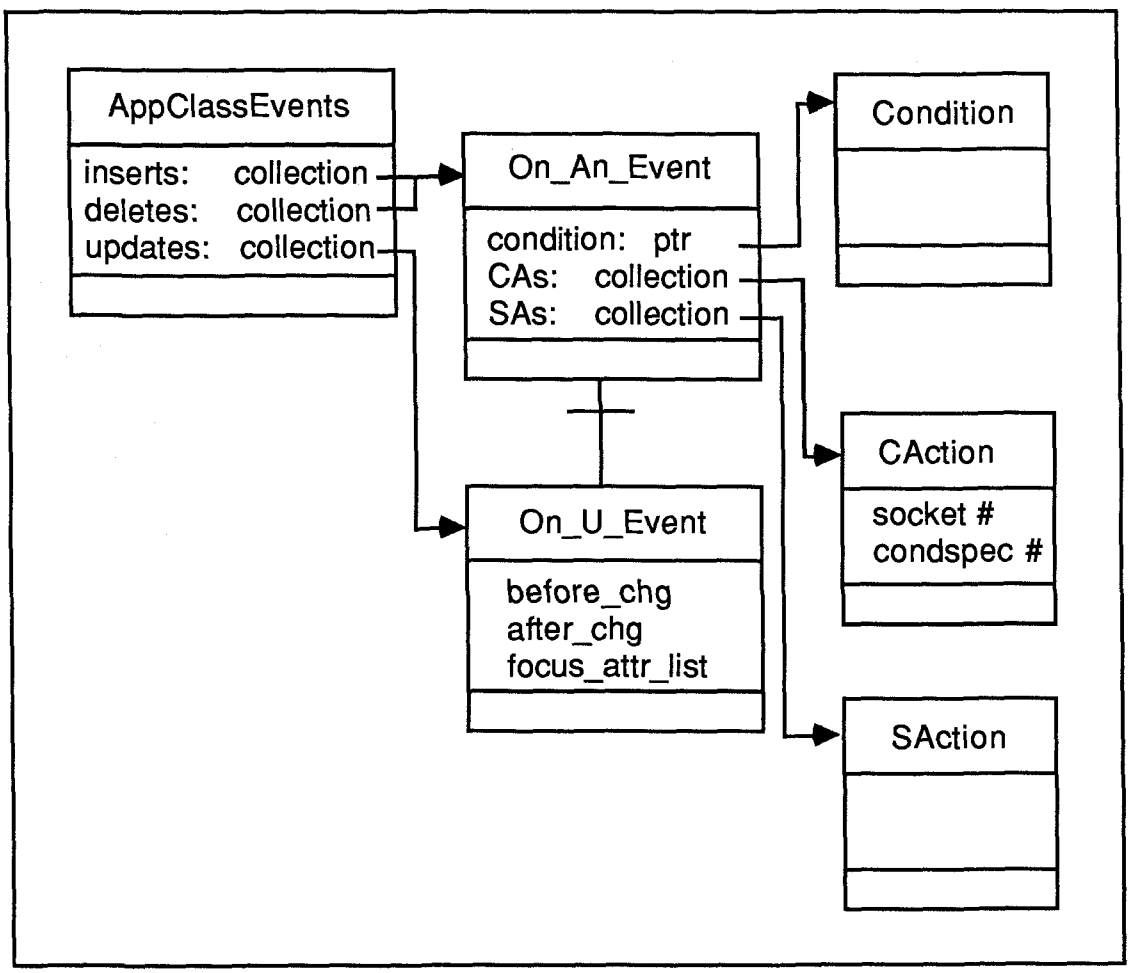



Figure 6-3 High-Level Monitored Event Set Structure

There is one AppClassEvents instance for each application schema class being monitored for change by the notification server. Each application class may have 0..n insert, delete, and update event conditions being monitored.

When an event occurs, and a Condition (the contents of a condition specification's WHERE clause) is true, the server responds by sending notification to one or more clients (using CAction information) and may also update its partial views (using SAction information).

In On_U_Event instances, the attributes before_chg and after_chg are boolean values which indicate whether the condition must be true before and/or after an object update. The following table shows the valid combinations, and what they represent:

| before_chg | after_chg | |
|---|---|---|
| True | True | object update |
| True | False | logical delete |
| False | True | logical insert |

On_U_Event instances also contain a list of 0..n "focus attributes" whose change of value is to be reported. For example, the condition specification below has two focus attributes: locn, and cur_speed.

[] ON U TO Freight_Train(locn,cur_speed) WHERE train_id = 1

The following is an example of how the monitored event data structure is used. The condition specification:

[1] ON I to Track_Segment() WHERE status = "closed"

asks that a client be notified when a Track_Segment object is inserted into the result; that is, when a new Track_Segment object with status = "closed" is created, or when an existing Track_Segment object has its status changed to "closed". Thus, the notification server has to set up the monitored event set to check each Track_Segment insert or update DB event. The corresponding monitored event set must contain the following information:

| Class & Events | Condition | Client Action | Server Action | Attribute List |
|---|---|---|---|---|
| Track_Segment | | | | |
| (insert) | status="closed" | (<socket#>,1) | () | |
| (update) | status="closed",F,T | (<socket#>,1) | () | () |

### 6.1.2.2 Intra-Object Condition Specifications

The condition specification [1] above is an intra-object [GJ91] example. These kinds of condition specifications are relatively straightforward to map into the monitored event set because they contain no joins (not even self-joins). All the information necessary to determine if a change has occurred to the result can be found in a single DB update event message. Other intra-object examples are:

[2] ON D TO Train*() WHERE cur_speed > speed_lim

[3] ON U TO Freight_Train(speed_lim) WHERE train_id=1

[4] ON U TO Train_Schedule!() WHERE train_id = 1

[5] ON U TO Track_Segment(status) WHERE in_view(West,seg_id)

Example [2] uses an inheritance hierarchy reference. The notification server must generate two intra-object condition specifications from this as follows:

[2a] ON D TO Passenger_Train() WHERE cur_speed > speed_lim

[2b] ON D TO Freight_Train() WHERE cur_speed > speed_lim

When an existing Train* object has either cur_speed or speed_lim changed so that cur_speed is no longer greater than speed_lim, or when a Train* object with cur_speed greater than speed_lim is deleted, client notification will occur. The notification server has to check each Passenger_Train and Freight_Train update and delete DB event message.

For example [3], the notification server has to check each Freight_Train update DB event message. If the train_id = 1 and the speed_lim variable has changed, client notification will occur.

Example [4] uses an aggregation hierarchy reference. The notification server must generate two intra-object condition specifications from this as follows:

[4a] ON U TO Train_Schedule() WHERE train_id = 1

[4b] ON IUD TO Tsched_Segment() WHERE parent->train_id = 1

[4b] comes from the Train_Schedule attribute, segs, being a collection of pointers to Tsched_Segment objects - every attribute of the parent that is a pointer or collection of pointers will cause the server to monitor the object(s) the attribute(s) point to. The definition of Tsched_Segment should be checked for pointer attributes as well, to discover if the aggregation hierarchy extends any further. In this example, the hierarchy is only two levels deep.

Example [4] is an intra-object condition specification because (1) the WHERE clause references only the primary key, (2) Tsched_Segment objects have a parent attribute, and (3) the aggregation hierarchy is only two levels deep. These three properties ensure that all the information necessary to determine if a change has occurred to the result of [4b] can be found in a single DB update event message, because the parent attribute in a Tsched_Segment after-change image will contain the parent's key value (as discussed in Section 5.2.1).

Not all aggregation hierarchy condition specifications are the intra-object kind. See Section 6.1.2.4 below for a more detailed discussion.

Example [5] contains a function, 'in_view', which restricts the condition specification result to those Track_Segments in the western region. We assume here that the definition of 'West' is fixed; the notification server may wish to retrieve and store the definition of 'West', but there is no need to monitor it for change. The server just has to check each Track_Segment update DB event message. If it is in the western region, and the status variable value has changed, client notification will occur.

Example [5] shows that the notification server needs some knowledge of the functions which may be used in the condition specification WHERE clause. The content of the Function KB is left for future work.

The monitored event set will contain the following information after condition specifications [2], [3], [4] and [5] have been processed:

| Class & Events | Condition | Client Action | Server Action | Attribute List |
|---|---|---|---|---|
| Freight_Train | | | | |
| (delete) | cur_speed>speed_lim | (<socket#>,2) | () | |
| (update) | cur_speed>speed_lim,T,F | (<socket#>,2) | () | () |
| (update) | train_id=1,T,T | (<socket#>,3) | () | (speed_lim) |
| Passenger_Train | | | | |
| (delete) | cur_speed>speed_lim | (<socket#>,2) | () | |
| (update) | cur_speed>speed_lim,T,F | (<socket#>,2) | () | () |
| Track_Segment | | | | |
| (update) | in_view(West,seg_id),T,T | (<socket#>,5) | () | (status) |
| Train_Schedule | | | | |
| (update) | train_id = 1,T,T | (<socket#>,4) | () | () |
| Tsched_Segment | | | | |
| (insert) | parent->train_id=1 | (<socket#>,4) | () | |
| (delete) | parent->train_id=1 | (<socket#>,4) | () | |
| (update) | parent->train_id=1,T,T | (<socket#>,4) | () | () |

### 6.1.2.3 Inter-Object Condition Specifications

Inter-object [GJ91] condition specifications are the other case. They are more difficult to monitor because determining if the result has changed involves the examination of more than just a DB update event message. For example, the following condition specification:

[6] ON I TO Tsched_Segment() WHERE

Tsched_Segment.seg_id = Track_Segment.segment_id AND

Track_Segment.status = "closed"

asks that a client be notified if any train's schedule is affected by a track closure. Two sets of objects, Tsched_Segments and Track_Segments, are being joined. The result of this condition specification will change if a new or existing Track_Segment's status changes to "closed" and any Train_Schedule contains that segment, or if a "closed" track segment is added to any Train_Schedule's segment collection.

If we were to evaluate [6] as though it were a query, we would use a construction diagram [BC79] like the following:

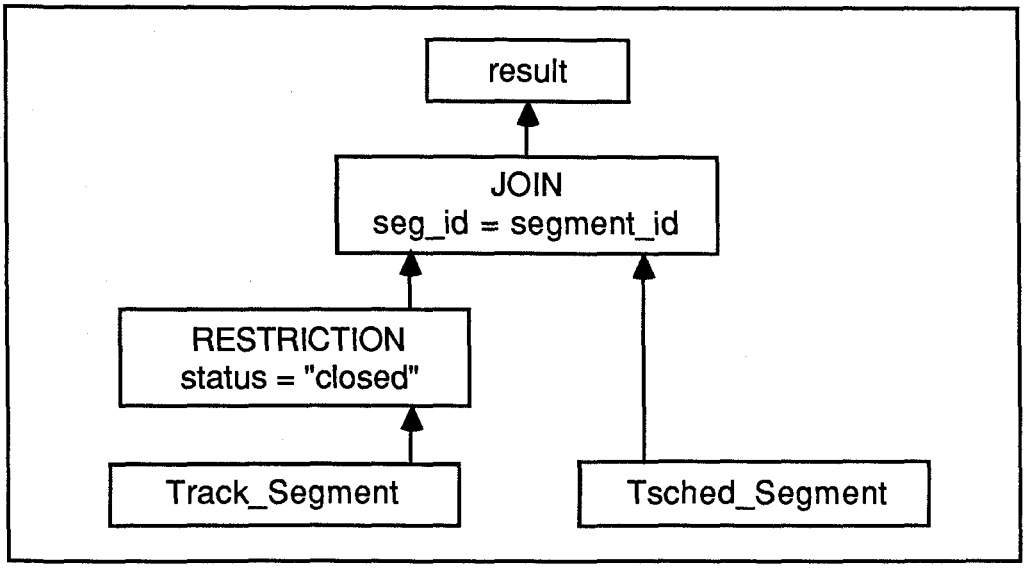



Figure 6-4 Query Construction Diagram

First, a partial result (or view) would be generated: the restriction of all Track_Segments with status = "closed". Then that partial result would be joined with all Tsched_Segments to determine the final query result.

From the above, we can determine what the notification server has to do to monitor condition specification [6] for change. First, it must maintain two partial views because the notification server is BEHIND in its processing of DB change. The partial views represent the state of the DB at the time of the DB update; if the server reads data from the DB instead of using the partial views, it will assess change based on a future DB state and may then be incorrect in its condition evaluation.

For example [6], the partial views are:

. view1: the collection of all Track_Segment objects with status = "closed"

. view2: the collection of all Tsched_Segment objects

Second, the server must set up the monitored event set as follows:

| Class & Events | Condition | Client Action | Server Action | Attribute List |
|---|---|---|---|---|
| Track_Segment | | | | |
| (insert) | status="closed" | (<socket#>,6) | (see A1) | |
| (update) | status="closed",F,T | (<socket#>,6) | (see A1) | () |
| (update) | status="closed",T,F | (<socket#>,6) | (see A2) | () |
| (delete) | status="closed" | () | (see A2) | |
| TSched_Segment | | | | |
| (insert) | | (<socket#>,6) | (see A3) | |
| (update) | | () | (see A4) | () |
| (delete) | | () | (see A5) | |

continued on next page

```
A1:   add the object to partial view1.  If the object's
      seg_id matches any segment_id in view2, notify clients

A2:   remove the object from partial view1

A3:   add the object to partial view2.  If the object's
      segment_id matches any seg_id in view1, notify clients

A4:   update the object in partial view2.  The segment_id is
      a primary key field, and an update event will not
      change its value (delete/add to change key).  The
      server could ignore UPDATES to partial view2 as they
      do not cause the kind of change [6] is interested in,
      but the server may be using partial view2 for other
      condition specifications, so it is best to keep view2
      up-to-date.

A5:   remove the object from partial view2
```

#### 6.1.2.4 Aggregation Hierarchy Condition Specifications

Example [4] above illustrated an aggregation hierarchy condition specification that was intra-object because of its three properties: (1) the WHERE clause referenced only the primary key, (2) there was a 2-way navigational reference (i.e., child objects had an attribute which pointed to the parent object), and (3) the aggregation hierarchy was only two levels deep. When one or more of these properties do not hold, at least part of the process of mapping the original condition specification into the monitored event set involves an inter-object condition.

For example, the notification server will split the following condition specification (which does not reference a key field in the WHERE clause):

[7] ON UID TO Document_Hdr!() WHERE author = "Kathy"

into one intra-object condition specification, [7a], and one inter-object condition specification, [7b]:

[7a] ON UID TO Document_Hdr() WHERE author = "Kathy"

[7b] ON UID TO Paragraph() WHERE parent->author = "Kathy"

The server must keep a partial view of all Document_Hdr objects where author = "Kathy", and then whenever a Paragraph object changes, check to see if the parent's key value is found in the partial view - if it is, then client notification occurs.

If Paragraph objects do not contain a parent attribute (no 2-way navigational reference), the server has no way to connect a Paragraph update event with its parent, so it must keep partial views. To reduce the number of different cases that the notification server must handle, we can make it a rule that 2-way navigational references must be added to application DB schema for notification purposes if they are not already defined.

If aggregation hierarchies are more than two levels deep, the notification server must keep at least one partial view for each level beyond 2. For example, let's say we have a class A which contains an attribute which points to objects of class B, and class B contains an attribute which points to objects of class C - a three level aggregation hierarchy. The notification server receives the following condition specification

[8] ON UID TO A!() WHERE key = k

and splits it into two intra-object condition specifications, [8a] and [8b], and one inter-object condition specification, [8c]:

[8a] ON UID TO A()

[8b] ON UID TO B() WHERE parent->key = k

[8c] ON UID TO C() WHERE parent->parent->key = k

The notification server must keep a partial view of all objects of class B whose parent->key = k so that when an object of class C changes, the server can check to see if its parent is found in the partial view.

## 6.2 HIGH-LEVEL PROCESSING ALGORITHMS

This section describes the notification server's high-level processing algorithms. Figure 6-5 shows the functions we will describe and their calling structure.

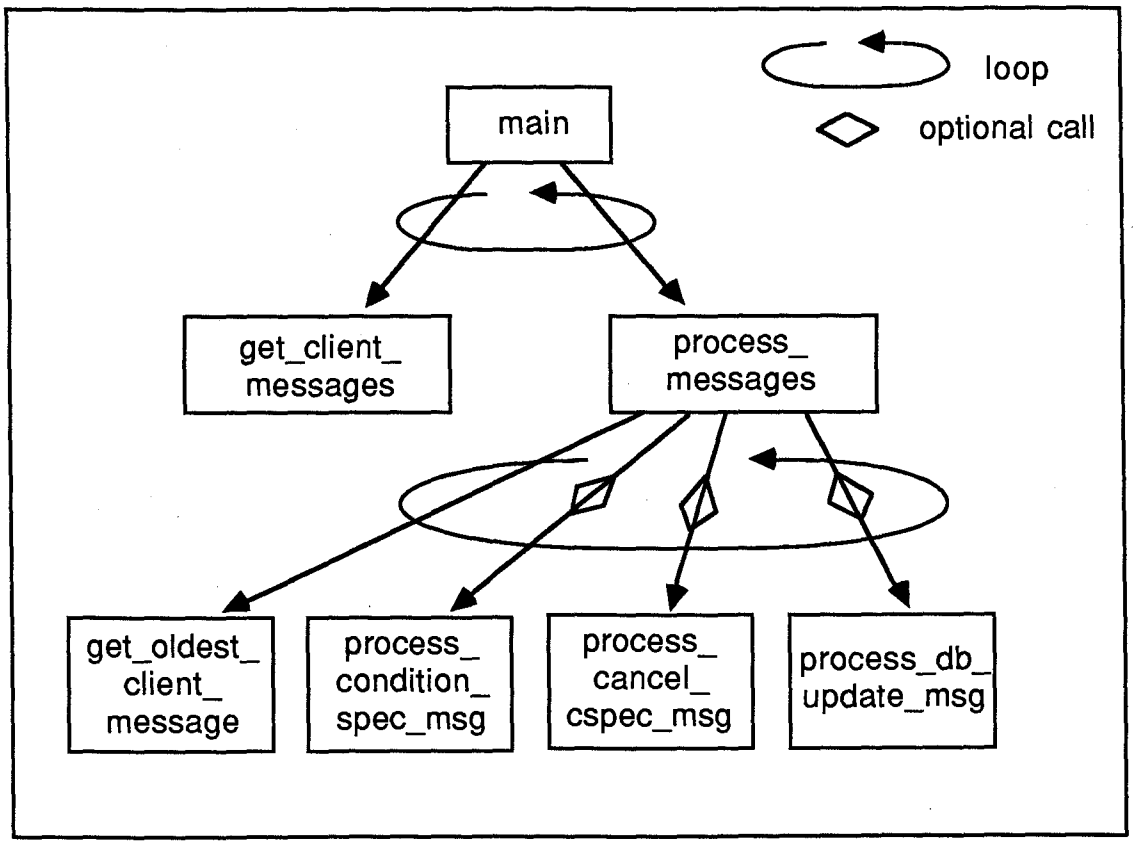



Figure 6-5 Server High-Level Function Hierarchy Chart

```
BEGIN main

    create socket(S0) where clients establish connection
    read application DB schema KB and function KB
    initialize monitored event set

    while(not done)

     if socket S0 has a message pending
       add new client (with socket Sn) to monitored event set

     if any client sockets (S1..Sn) have messages pending
       MsgList = get_client_messages
       process_messages(MsgList)

    END while

  END main
```

```
BEGIN get_client_messages

 read the first message from each client socket that has a message
  and append it to MsgList

 return MsgList

END get_client_messages
```

```
BEGIN process_messages(MsgList)

 while (get_oldest_client_message(MsgList,Msg))

  CASE Msg.msg_type OF

    condition_spec_msg:
      process_condition_spec_msg(Msg)

    db_update_event_msg:
      {
       OMsg = Msg
       while ((Msg.msg_type == db_update_event_msg) AND
              (Msg.timestamp == OMsg.timestamp))

           process_db_update_msg(Msg)

           NMsg = read next message from the client socket

           if (another message from client found)
             if ((NMsg.msg_type == db_update_event_msg) AND
                 (NMsg.timestamp == OMsg.timestamp))
                   OMsg = Msg
(A)                Msg = NMsg
             else
                   undo socket read (put NMsg back on queue)
                   Msg.msg_type = no_msg
           else
              Msg.msg_type = no_msg

        END while
      }

    cancel_cspec_msg:
      process_cancel_cspec_msg(Msg)

  END case
  END while

END process_messages
```

(A) all DB update event messages from the same transaction will have the same timestamp, and they should be processed one after another.

```
BEGIN process_condition_spec_msg(Msg)

  parse the message to validate and interpret it

  if (valid_message)
    if partial view(s) must be maintained
       read the data from the database
    send acknowledge message to client (valid/active)
    add information to the monitored event set
  else
    send acknowledge message to client (error)

END process_condition_spec_msg
```

```
BEGIN process_cancel_cspec_msg(Msg)

  CASE Msg.cancel_flag OF

       'A' (all):
         remove all events corresponding to a client's condition specifications
           from the monitored event set

       '1' (one):
         remove all the events corresponding to the particular condition
           specification from the monitored event set

  END case

END process_cancel_cspec_msg(Msg)
```

```
BEGIN process_db_update_msg(Msg)

  if (Msg.class_name in monitored event set)
   if (Msg.event in monitored event set for class)
        CASE Msg.event OF
              'I','D':
                if (a condition in the monitored event set
                    is TRUE for Msg.after_image)
                       send notification message to client(s)
                       perform the appropriate server actions (if any)

              'U':
                {
                ac = the value of a condition in the monitored event set,
                       using Msg.after_image
                bc = the value of a condition in the monitored event set
                       using Msg.before_values

                if (ac == after_chg value of the condition as specified in
                           the monitored event set) AND
                   (bc == before_chg value of the condition as specified in
                           the monitored event set)  AND
                    (either there are no focus attributes, or at least one focus
                     attribute is found in Msg.before_values

                     send notification message to client(s)
                     perform the appropriate server actions (if any)
                }

        END case

END process_db_update_msg(Msg)
```

# CHAPTER 7
# Internal Design of Client Processes

In this chapter, we describe how client processes send DB update event messages, and how clients might be designed to receive and process notification messages.

It is not our intention here to completely specify a client's detailed design, since much of each client process is application-specific. Instead, we only wish to show how an implementation of the notification functionality might be done.

## 7.1 SENDING DB UPDATE EVENT MESSAGES

Methods in the DB class schema which perform database updates could include code to send message(s) to the notification server to signal DB update event(s). For example, here is a partial definition of the Train class:

```
class Train  {
    int   train_id;
    int   cur_speed;

  public:

    Train (int t, int s)  {  // constructor
        train_id = t;
        cur_speed = s;
        get_cur_time(t);
        send_db_update_message(t,Train,...);
       }
    ...
   };
```

Then, using code like the following example, a client process can create a persistent Train instance in the ObjectStore database:

```
Train  *a_train;
do_transaction(0, transaction::update)  {
    a_train = new(db)  Train(1,50);
   }   //  commit point
```

Unfortunately, the placement of message-sending function calls in class methods isn't appropriate for two reasons:

(1) Since the Train class constructor method (and any other method accessing persistent data) must be invoked from within a transaction, DB update event messages are sent to the notification server BEFORE commit. The possibility of transaction aborts, nested transactions, and multiple invocations of the same method within a transaction severely complicates the design of a notification server that receives uncommitted update event messages. Therefore, this thesis restricts DB update event messages to COMMITTED changes.

(2) if a client creates a transient instance of the Train class (a local object), a DB update event message will be sent to the notification server! Persistence is not part of an object's type - the Train object itself does not, and should not, care where it is stored. We could get around this problem by passing a flag to every class method which updates the object to indicate whether or not the notification server should be sent a message, but that makes the code more awkward.

So, in our approach DB schema class methods do not contain code to send messages to the server; instead, clients send update event message(s) to the server after changes have been committed:

```
Train  *a_train;

do_transaction(0, transaction::update)  {

    a_train = new(db)  Train(1,50);

    get_cur_time(t);

   }   //  commit point

send_db_update_message(t,Train, ... );
```

This approach guarantees that only committed change information is sent to the server, but it has a weakness. If a client processes terminates after committing a change, but before it has a chance to send an update message to the server, notification is lost. The server can detect that a client process has terminated, but has no idea what the client might have done to the DB just prior to termination. The server's only recourse is to notify all other clients that notification of change may have been lost. Clients can then choose to poll the DB to check for change.

A client may make changes to several objects in one transaction. For example:

```
Train           *a_train;
Train_Schedule  *a_tsched;

do_transaction(0, transaction::update)  {

   a_train  = new(db)  Train(1,50);
   a_tsched = new(db)  Train_Schedule(...);

   get_cur_time(t);

  }   //  commit point

send_db_update_message(t,Train,...);
send_db_update_message(t,Train_Schedule,...);
```

All DB update event messages for one transaction must carry the same timestamp. This ensures that the notification server will process all messages from one transaction before doing anything else. We assume that a client process cannot have multiple independent transactions in progress at the same time, but can nest transactions within one another. Client processes can send DB update event messages for a nested transaction as soon as the commit point for that transaction has been reached (which may not be until the outermost transaction commits, depending on the implementation of the underlying OODBMS).

## 7.2 MAIN PROCESSING ALGORITHMS

A client process may, or may not, use a window management system. We have considered both cases in the sections which follow.

The algorithms below show all the client's condition specifications being sent to the server, one after another, near the beginning of the program. This is a simplification for illustration purposes only - client processes can send, and cancel, condition specifications at any time.

### 7.2.1 Main Body Without a Window Management System

If a client process does not use a window management system, we assume that a child process is forked to wait for messages from the notification server so that the client (the parent) is free to perform other processing (see illustration in Figure 5-1). The child uses a signal to interrupt the parent when a message arrives. The parent's signal handler function reads and processes the message (or messages), then sends a signal back to the child to start it watching again. The software designer must remember to disable signal interrupts during critical sections of code.

```
BEGIN main

register a message_handler function, to be invoked when this client
 process receives a signal from its child indicating that a message has
 arrived from the server

create a socket(Cn) and connect to server

fork a child to listen for messages from server

send condition specification(s) to server

wait for acknowledgement from server that each condition specification
 sent has been accepted and is now active

while (not_done)
   do client processing
END while

cancel condition specifications
close socket

END main
```

### 7.2.2 Main Body With a Window Management System

If a client process uses a window management system, it must usually allow the window system to handle all its inputs, including the arrival of messages at a socket. The "notifier model" of input handling is typical of toolkits like SunView or Xviews [FVFH90] - we use this model as our example.

The client process registers a function with a central Notifier for each input event (e.g., keypress, mouse movement, message arrives at a socket) the client wishes to handle. The Notifier can either be a separate process or a procedure linked in with the client's code. The client turns over control to the Notifier. When an input event occurs, the Notifier will "call-back" the appropriate function registered by the client. Figure 7-1 [Sun90] illustrates the flow of control in a Notifier-based client process.

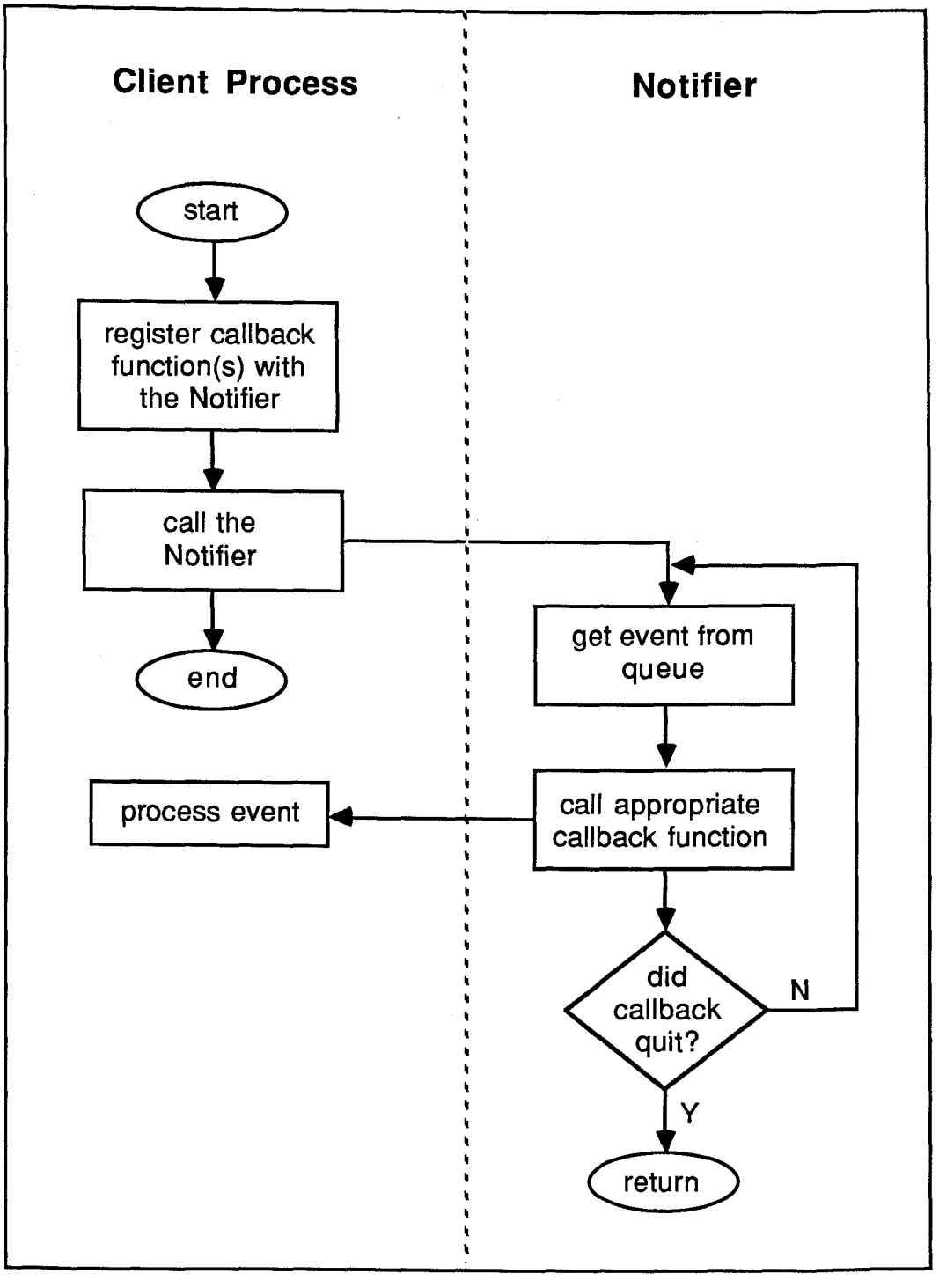



Figure 7-1 Flow of Control in a Notifier-Based Client

The Notifier has a default order (a priority scheme) for calling back event handler functions. It is important for the software designer to determine the Notifier's priority for

socket events - if it is too low, there may be times when notification messages are delayed by client input activity rather than by the performance of the notification server.

---

**BEGIN main**

create a socket(Cn) and connect to server

send condition specification(s) to server

wait for acknowledgement from server that each condition specification sent has been accepted and is now active

initialize the window environment

register the socket(Cn) and a message_handler function with the Notifier, to watch for messages from the notification server

hand over control to the Notifier, which will call back a client whenever an input event occurs

cancel condition specifications
close socket

**END main**

---

### 7.2.3 Processing Messages from the Server

Regardless of whether the message_handler function is invoked when a signal arrives from a child process, or when a window system Notifier calls back the function, the same basic processing is done by the function.

A client can receive two message types: condition specification acknowledge messages, and change notification messages (see the high-level description given below). Just what the client does when it receives these messages is application-dependent. For acknowledge messages, it may set global variables which indicate the status of condition specifications sent to the server. For notification messages, it may update local data copies and/or refresh screen displays.

```
BEGIN message_handler

 read message from socket(Cn)

 CASE msg_type OF

       acknowledge_msg:
        {
        CASE status OF

              valid:  ...
              error:  ...

        END case
        }

       notification_msg:
        {
        CASE condition_id OF

              1: ...
              2: ...          } one case for each condition
                                specification sent to the server
              ...

        END case
        }

 END case

END message_handler
```

# CHAPTER 8
# Extending the Design to Handle Multiple Versions

In the previous chapters we have assumed that there was only one version of each database object. Whenever an object update occurred, the new attribute value(s) overwrote the previous one(s).

This chapter discusses how our change notification design could be extended to handle the versioning of objects in application DB schemata. We assume that the underlying OODBMS supports versioned data through some form of embedded version management scheme.

First, we describe our model of version management. Then, we discuss how our change notification server complements an application where clients are versioning data. Finally, we describe the extensions which must be made to our design of the interface between DB clients and the notification server.

## 8.1 OUR MODEL OF VERSION MANAGEMENT

Our model is somewhat based on ObjectStore's version management mechanism [OD91a] [OD91b] [LLOW91], which in turn has been influenced in part by the model described in [CK88], but we have abstracted and simplified ObjectStore's approach to make our model much less implementation-dependent, much easier to understand, and much easier to design a change notification mechanism for.

For a comparison of our model with ObjectStore's version management scheme see Appendix A.

### 8.1.1 Basic Concepts

A version unit is a persistent instance of a class whose purpose is to group together and order a number of persistent objects called versions. Each version contains one or more data objects of one or more application classes. The ordered set of versions in a version unit records the modification history for the data objects.

Each version in a version unit has a unique identity and can be directly accessed. A version is updatable (i.e., its content can be overwritten) until a DB client decides that its current state should be saved, or "frozen", for all time. Once the first version of a version unit is frozen, change can only be made by first creating an updatable copy (a new version) of version 1. When the second version is frozen, a new version can be derived from it, and so on. The history of change forms a version-derivation hierarchy [CK88], or version graph [OD91a].

To simplify what our change notification design must handle, our version management model assumes that different version units do not overlap (i.e., a data object cannot be a member of more than one unit), and that objects in one version unit do not contain pointers to objects in another unit. These issues are left for future work.

The following example is presented to help clarify the basic version management concepts:

> Let's say we wish to store a number of project documents in the database (using the Document_Hdr and Paragraph classes as defined in Chapter 4). We could create a separate version unit for each document in the document set (option A below), or we

could create one version unit that contains all documents in the document set (option B below).

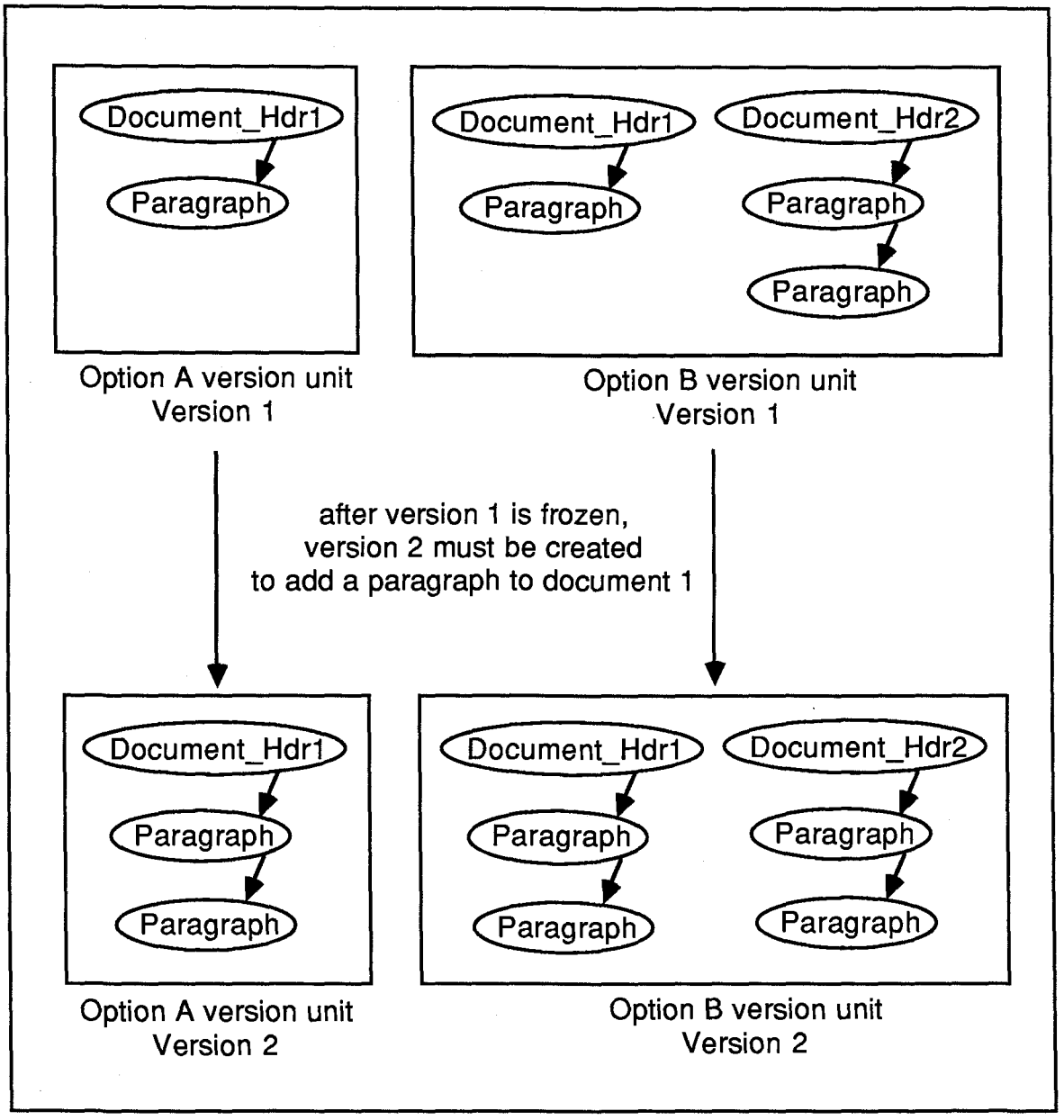



Figure 8-1 Illustration of the Basic Version Management Concepts

Which option we choose depends on the kind of history we wish to keep, on how clients intend to update documents, and on how related one document's content is to

another's. Option A would keep a separate history of each document, and would reduce conflict when concurrent clients work on different documents. If change to one document often means change to another, option B would allow us to save internally consistent versions of a set of documents, rather than having to determine which version of each option A document version unit makes a consistent set.

### 8.1.2 The Basic Version Management Model

The simplest version graph is linear - Figure 8-2 below provides an example.

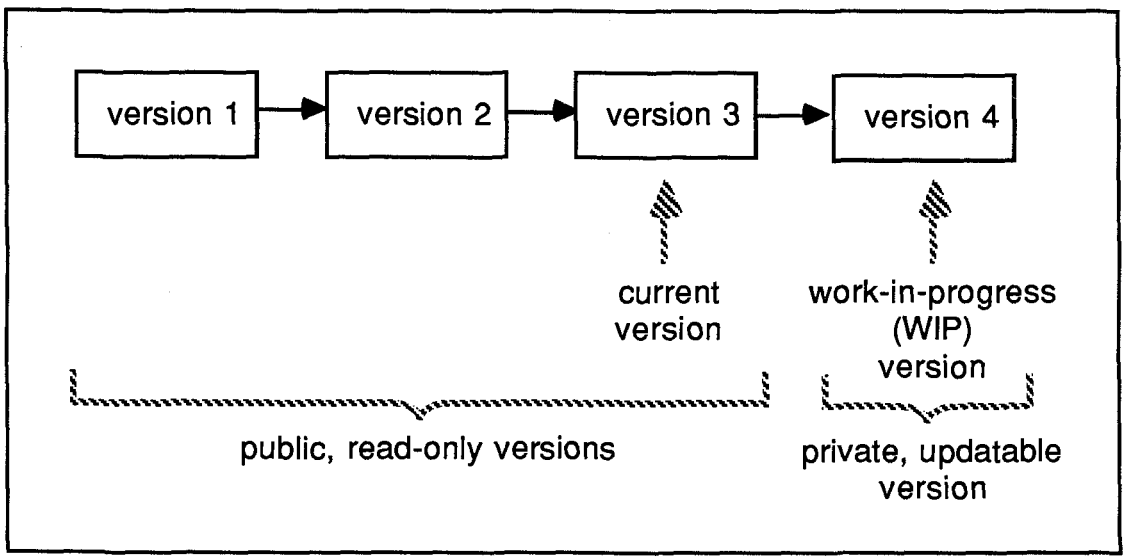



Figure 8-2 A Linear Version Graph

A version number is all that is necessary to uniquely identify a particular version in a linear version graph.

Versions 1, 2, and 3 are read_only versions (also referred to as "stable" or "frozen" versions) of the version unit . Version 3 is the latest read-only version, and is referred to as the current version. Version 4 is an updatable version, currently undergoing change.

Versions 1, 2, and 3 are public versions; that is, they can be read by any DB client. Version 4 is a private version, accessible only by the client who created it and is in the process of updating it.

Figure 8-3 lists the version-level access and manipulation functions required by our basic model.

```
new_version <= checkout (unit_name)
checkin (unit_name)
version <= read (unit_name, version_number)
version <= read_next (unit_name, version_number)
version <= read_prev (unit_name, version_number)
delete_unit (unit_name)
delete_version (unit_name), version_number)
```

Figure 8-3 Basic Version-Level Access and Manipulation Functions

The **checkout** function creates a new version (an updatable WIP version) by copying the current version. Only the client that checked out the current version can make changes (i.e., insert, update, delete objects) to the WIP version. A version cannot be checked out once it has a successor.

The **checkin** function makes the WIP version the current (latest public) version. The WIP version must be checked in by the client that checked it out.

The three read functions (**read**, **read_next**, **read_prev**) return the appropriate version of the specified version unit. If the version_number parameter is 0, then the current version is read, or is used as the reference node from which to determine the next or previous version. If the WIP version is the function result, it is only returned if the client

who called the function is the owner of the WIP version. Thus, other clients cannot even discover that a WIP version exists unless they call the **checkout** function and it is unsuccessful because the current version already has a successor.

An entire version unit (all versions) can be removed from persistent storage by using the **delete_unit** function. This kind of delete should not be allowed if the current version of the version unit is checked out.

A node can be removed from a version unit's version graph by using the **delete_version** function. A read-only version can only be deleted if (1) it has no parent, and (2) it is not currently checked out. The first restriction forces version history to be compressed (i.e., removing versions from oldest to newest). Deleting a version with an existing parent would be like "ripping out a chapter from a history book" - continuity would be lost.

If the **delete_version** function references a WIP version, the delete is, in effect, a rollback of a checkout operation. Only the owner of a WIP version can delete the WIP version.

### 8.1.3 Allowing Multiple Branches

Our model becomes more complex when the version graph is allowed to have branches, as in Figure 8-4 below.

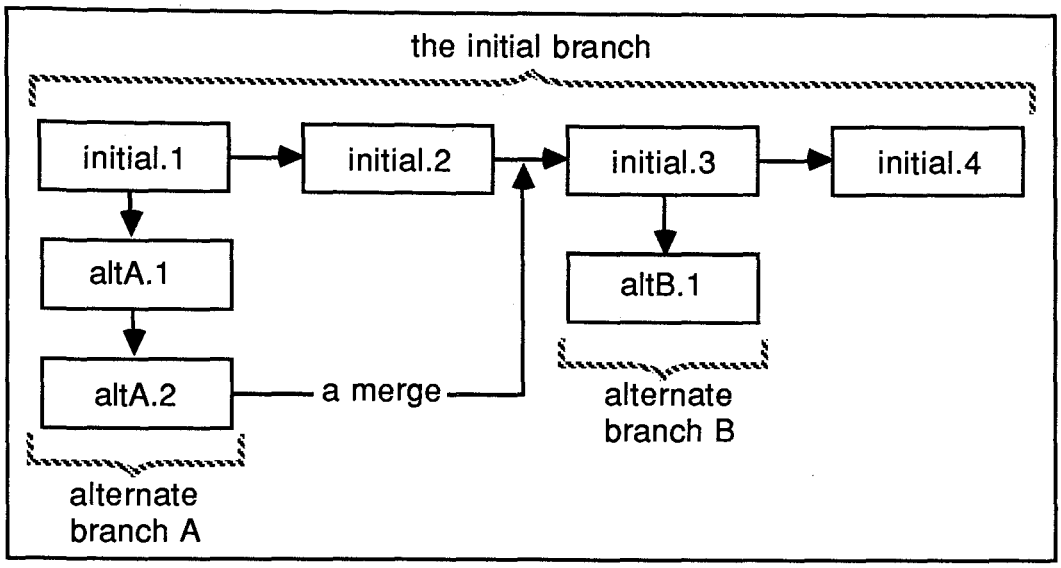



Figure 8-4 A Multiple-Branch Version Graph

Each version is uniquely identified by a name, 'branch_name.n'. The initial branch's name is "initial"; an alternate branch is given a name by a DB client when it is created. Versions are sequentially numbered within a branch.

An alternate branch is created when clients wish to make concurrent updates to a version unit (e.g., versions initial.2 and altA.1/altA.2 in Figure 8-4) or, generally, any time an in-line (same branch) successor already exists for a version. Alternate branches may be merged at a later time, or they may remain separate to represent diverging data (e.g., document sets for different customers). For simplicity, we will allow a node in the version graph to have at most two children: an in-line successor, and one alternate branch successor.

Figure 8-5 shows the version-level access and manipulation functions required by our model when multiple branches are allowed.

```
new_version <= checkout (unit_name,branch_name)

new_version <= checkout_branch (unit_name,
                                from_version_name,
                                new_branch_name)

checkin (unit_name,branch_name)

new_version <= merge (branch1_name,branch2_name)

version <= read (unit_name,version_name)

version <= read_next (unit_name,version_name)

version <= read_prev (unit_name,version_name)

version_list <= read_all_next (unit_name,version_name)

version_list <= read_all_prev (unit_name,version_name)

delete_unit (unit_name)

compress_versions (unit_name,to_version_name)

delete_WIP_version (unit_name,version_name)
```

Figure 8-5 Multiple-Branch Version-Level Access and Manipulation Functions

The **checkout** function creates a new version (an updatable WIP version) by copying the current version of the specified branch.

The **checkout_branch** function creates the first version (an updatable WIP version) for a new branch by copying the specified version. Only the client that checked out the specified version can make changes (insert, update, delete objects) to the WIP version. A new branch cannot be created from a version once it has one branch successor.

The **checkin** function makes the WIP version the current (public) version of the branch.

The **merge** function creates a new version (an updatable WIP version) by copying the current version of the first specified branch. It is then up to the client to merge the contents

of the current version of the second specified branch with the WIP version. The **checkin** of the WIP version makes it a read-only version which supersedes the current versions of both branches involved in the merge.

The first three read functions (**read**, **read_next**, **read_prev**) return the appropriate version of the specified version unit. If the version number part of the version_name parameter is 0, then the current version on the specified branch is read, or is used as the reference node from which to determine the next or previous version. If there is more than one next or previous version, the one on the same branch as the reference node is returned.

The two new read functions (**read_all_next**, **read_all_prev**) return a list of versions. These functions are used to determine if a node in the version graph has more than one child (alternate branches) or parent (a merge).

The **delete_unit** function is the same as described for our basic model.

Deleting a read-only version is more difficult now than in our basic model. A read-only version can only be deleted if (1) it has no parents, (2) the result does not orphan an alternate branch (see Figure 8-6 below), (3) merge continuity is not lost (see Figure 8-7 below), and (4) the version is not currently checked out.

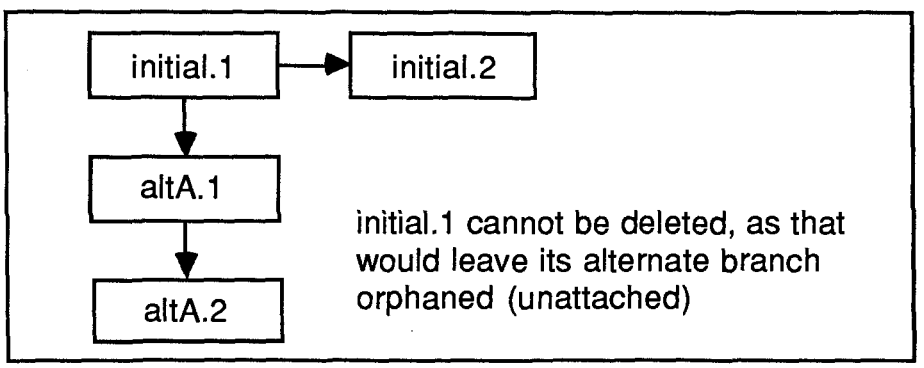



Figure 8-6 Orphaned Branch Problem

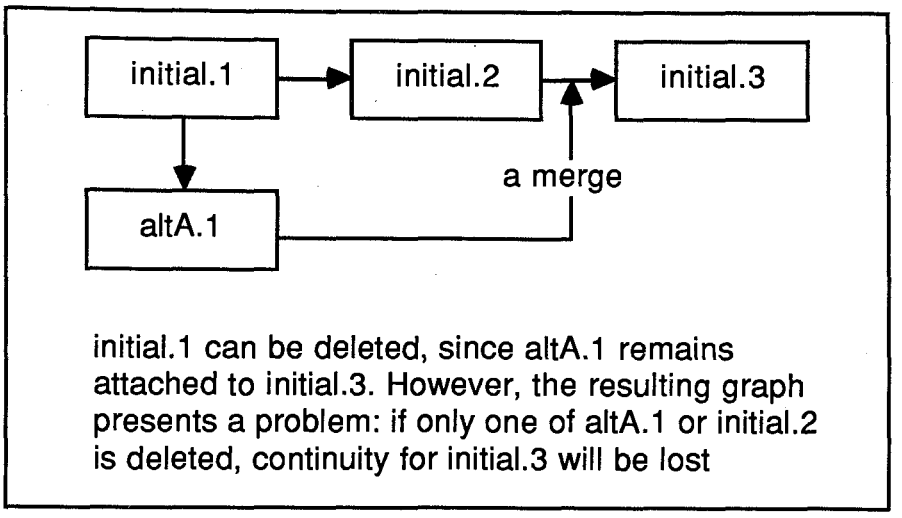



Figure 8-7 Merge Continuity Problem

Conditions (2) and (3) mean that some versions in a multiple-branch version unit cannot be deleted with the **`delete_version`** function. Therefore, we replace that function with a new function, **`compress_versions`**, where all previous versions up to, but not including, the referenced version are deleted as long as conditions (1) through (4) above can be met. Compressing up to version initial.2 in Figure 8-6 removes an entire branch; compressing up to version initial.3 in Figure 8-7 removes all evidence of the merge.

A new function, **`delete_WIP_version`**, provides the way to rollback a checkout operation on any branch. Only the owner of a WIP version can delete the WIP version.

### 8.1.4 Sharing the WIP Version

So far, we have assumed that a WIP version can only be read, updated, checked in, and deleted by its owner (the client that checked it out). This is too restrictive for some cooperative work, or "groupware", applications where the owner wishes to share by granting other clients access to the WIP version.

There are two dimensions to sharing: the possible degrees of sharing (i.e., read-only, read-write, read-write-delete-checkin), and which other clients are granted what degree of sharing (i.e., none, all, one, group(s)).

We extend our model to allow the WIP owner to grant read-only, read-write, or read-write-delete-checkin access to all or no other clients. We define a function, **change_WIP_access**, which allows only the WIP owner to broaden WIP access. Once public, a WIP version cannot be made private; however, the degree of public sharing can be changed at any time.

```
change_WIP_access (unit_name, version_name, new_access)
```

Figure 8-8 change_WIP_access Function

## 8.2 USING THE CHANGE NOTIFICATION SERVER

Cooperative work, or groupware, applications often require a DB schema which maintains multiple versions of data objects. Individuals, or groups of clients, need to work in parallel on the same data (i.e., creating alternate branches of a version unit) because either they are working on separate, but equally valid, versions of the data (e.g., alternate designs for a car), or they are circumventing the delay of another client's long-term update by creating their own version (with the intent to merge their version with another's at some later time).

Cooperative work applications require clients to be kept informed of what others are doing, not only to keep a local copy of a public WIP version up-to-date or to watch for particular data patterns in a public WIP version, but to follow the structural changes to version graphs (i.e., checkout, checkin, version delete, etc.). Clients could poll the database, searching for change in version graphs and change to data objects in WIP versions, but this places an undesirable I/O and processing burden on clients (especially when version graphs are large

and complex). Integrating a change notification server with knowledge of the underlying OODBMS version management scheme into a cooperative work application provides clients with an effective alternative to polling.

If we assume that a DB client only wishes to receive change notification for updates to the content of one public WIP version at a time, we can partition clients into groups, and then we can make the notification server even more supportive of cooperative work.

The notification server can partition clients into groups based on the WIP version they wish to receive notification for when that version's content is changed. Since each branch of a version unit can have at most one WIP version, clients select the WIP version, and thereby assign themselves to a group, by specifying a branch. If the branch doesn't currently have a WIP version, or the WIP version is currently private, clients are still members of the group (there just isn't anything to notify them of yet).

Once a client is assigned to a group, the condition specifications it sends to the notification server do not have to specify which WIP version to monitor for change; the WIP version is given by group membership. Thus, clients can change groups, or WIP versions can be created and checked in along the client's current branch, and the notification server can adjust the focus of a client's active condition specifications automatically. Clients do not have to cancel and re-issue condition specifications each time they change groups or a new WIP version is created.

In addition to providing flexible condition specification, the notification server's group membership information can be provided to clients upon request in order to, for example, enable clients to communicate directly with other group members to coordinate updates to a public WIP version.

## 8.3 CLIENT/SERVER COMMUNICATION REVISITED

To review, clients send one or more condition specifications to the notification server to indicate what DB changes they wish to be notified of, and they send a DB update event message to the notification server for each change made to application data after the change has been committed to the OODBMS. The notification server sends an acknowledge message to clients for each condition specification it receives, and the server sends a notification message to clients for each change they have indicated an interest in. The general system architecture is shown in Figure 8-9 below.

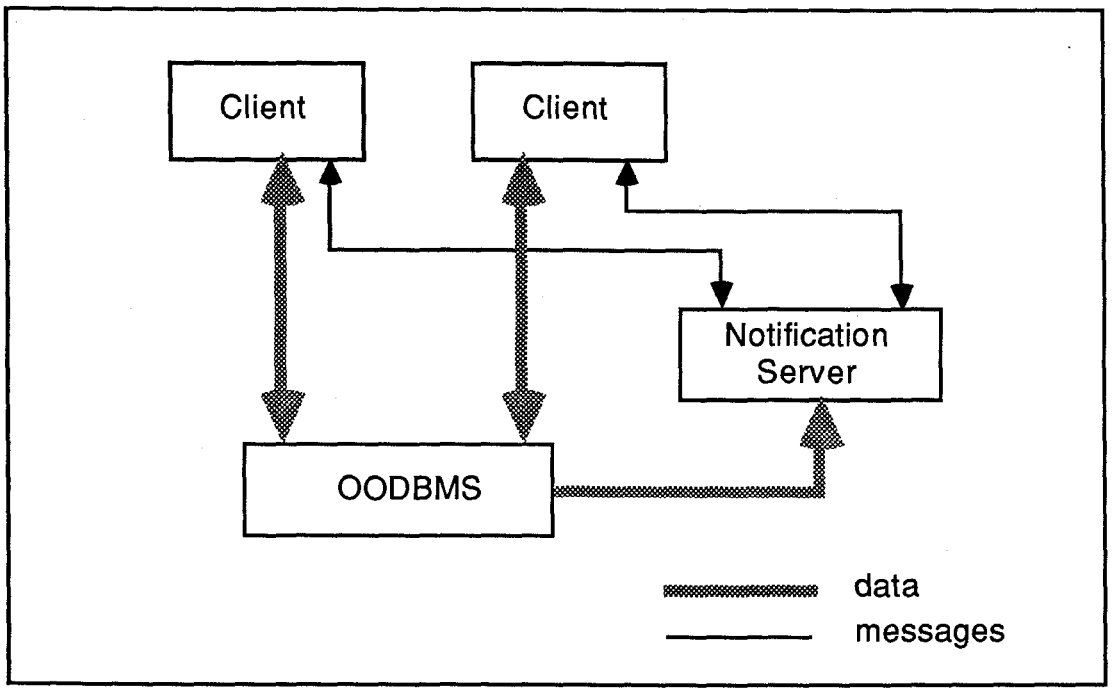



Figure 8-9 General System Architecture (Revisited)

In previous chapters, our notification server design addressed single-version data-object-level DB update events, condition specification, and change notification.

When an application's DB schema includes version units and versions, the design of data-object-level communication between clients and the notification server remains relatively unchanged (there are small modifications to the content of some messages), and the full

scope of data-object-level condition specification is still possible (though implementation may be more complex in some cases).

Data-object-level updates are made to public WIP versions by one or more clients. Data-object-level notification allows clients sharing the WIP version to keep local copies of the data up-to-date, and/or watch for special conditions (e.g., a new document added to a document set).

Since data objects can now be contained within version units and versions, a second level of database change must be addressed. Version-level DB update event messages, condition specification, and change notification messages must be added to the server design.

Version-level updates are changes made to a unit's version graph or to a version's public/private access. Notification of these changes can tell a DB client that, for example, the current version on a particular branch has been checked out, or a WIP version has been checked in, or a WIP version now has public access.

Messages sent by clients to the server to request a group change or request a list of group members, and messages sent by the server to clients to acknowledge a group change or return a list of group members, must also be added to the server design.

A list of all the messages which may now be passed between clients and the notification server is given in Table 8-1 below. Each of these messages is discussed in more detail in the sections which follow.

| FROM CLIENTS TO THE SERVER | FROM THE SERVER TO CLIENTS |
|---|---|
| Version-level DB update events<br>Data-object-level DB update events | Version-level Notification<br>Data-object-level Notification |
| Version-level Condition Specification<br>Data-object-level Condition Specification | Acknowledge Condition Specification |
| Cancel Condition Specification | |
| Group Change Request<br>Group List Request | Acknowledge Group Change<br>Group List Reply |

Table 8-1 Messages Passed between Clients and the Notification Server

### 8.3.1 Database Update Event Messages

#### 8.3.1.1 Version-Level Event Messages

The following are the version-level DB update events:

. checkout, checkout_branch
. merge
. checkin
. delete_unit, compress_version
. delete_WIP_version
. change_WIP_access

When the **checkout** or **checkout_branch** function is called, the client must send one message to the server: an 'update' for the version that was checked out.

When the **merge** function is called, the client must send two messages to the server: one 'update' for each version involved in the merge.

When the **checkin** function is called, the client must send one message to the server: an 'insert' for the WIP version if it remained private, or an 'update' for the WIP version if it was public.

When the **delete_unit** function is called, the client must send one message to the server: a 'delete' for the entire version unit.

When the **compress_versions** function is called, the client must send one message to the server: a 'delete' for all versions prior to the specified version.

When the **delete_WIP_version** function is called, the client is deleting a WIP version and rolling back a checkout, checkout_branch, or merge operation. If the WIP version is private, the client must send only an 'update' message to the server for each public version whose checkout or merge is being cancelled. If the WIP version is public, the client must also send a 'delete' message to the server for the WIP version.

When the **change_WIP_access** function is called, the client must send one message to the server: an 'insert' for the WIP version when its access goes from private to public, or an 'update' for the WIP version when its public access is changed (e.g., from read-only to read-write, etc.).

The notification server design is extended to include a version-level DB update event message sent by clients to the server. The message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'V' (version update message code) |
| Timestamp | commit time |
| Message length | # bytes |
| 'I' \| 'D' \| 'U' | Insert,Delete,or Update operation |
| Unit name | unique version unit name |
| if 'I' operation: | |
| version ID | branch name, version # |
| if 'D' operation: | |
| delete type | U (unit), W (delete WIP)<br>V (compress versions) |
| version ID | if delete type = 'V' or 'W',<br>branch name, version # |
| if 'U' operation: | |
| update type | CO (checkout), CB (checkout branch),<br>co (cancel checkout),<br>cb (cancel checkout branch),<br>MG (merge), mg (cancel merge),<br>CI (checkin), CA (chg WIP access) |
| version1 ID | branch name, version # |
| version2 ID | if 'MG', second version involved |
| new access | if 'CA', new WIP access for others |

Table 8-2 Version-Level DB Update Event Message Format

### 8.3.1.2 Data-Object-Level Event Messages

When a WIP version is public (i.e., access is read-only, read-write, or read-write-checkin-delete) clients must send a data-object-level update event message to the notification server for each data object insert, update, and delete they perform to that version.

Data-object-level events for a WIP version are the same as the DB update events for non-versioned data discussed in Section 3.1.1.

The data-object-level DB update event message format given previously in Table 5-2 must be modified (see Table 8-3 below) to handle the reporting of change to both versioned and non-versioned data. One new field, version unit name, has been added to the message so that the notification server can determine if and where the object is being versioned. Note that the WIP version name does not have to be part of the message because the notification

server knows which group each client belongs to, and each group can only be updating one WIP version of a version unit at a time.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'U' (DB update event message code |
| Timestamp | commit time |
| Message length | # bytes |
| 'I' I'D' I 'U' | object insert, delete, or update flag |
| Class name | object's class |
| Unit name | unique version unit name ** |
| After-change attribute values | value of each object attribute (ordered as listed in the class definition) |
| Attribute name * | if change flag = 'U', full name of the attribute that changed |
| Before value * | if change flag = 'U', value of the attribute before change |

* field repeated for each attribute modified
** field only used for versioned data; otherwise blank

Table 8-3 Extended Object-Level DB Update Event Message Format

### 8.3.2 Condition Specifications

#### 8.3.2.1 Version-Level Condition Specification

In this section we present a condition specification language that allows DB clients to describe a collection of public versions in a version unit that they wish the notification server to monitor for change.

The general format of the version-level condition specification language is as follows:

```
[ID#] ON <Chg_type> TO <unit_name>(<Chg_level>)
      WHERE <Predicates>  DO <Action>
```

Figure 8-10 Version-Level Condition Specification Language Format

The default condition specification query result is a collection of ALL public versions of a version unit. For example, the condition specification

[1] ON IUD TO proj1_doc_set()

asks for notification of any version-level change to any public version of the proj1_doc_set unit.

Remember that WIP versions of a version unit are private when created, so if the owner does not make a WIP public, clients who have issued [1] above will only receive notification of WIP change when a WIP version is checked in. If and when the owner makes a WIP version public, notification of that access change and any subsequent version-level changes to the WIP version will occur for clients who have issued a condition specification like [1] above.

Version-level condition specifications also cause the notification server to set the client's WIP group affiliation for the specified version. For example [1] above, the server sets the client's group to the default branch, 'initial'.

Using a keyword in the WHERE clause, version-level change notification can be restricted to a particular branch, and the server will set the client's WIP group affiliation to that branch. For example,

[2] ON UID TO proj1_doc_set() WHERE Branch_Name = "altA"

To receive notification of ALL object-level change to the public WIP version of the client's current group, a version-level condition specification can include the chg_level keyword 'OLEVEL'. For example,

[3] ON UID TO proj1_doc_set(OLEVEL)

[4] ON UID TO proj1_doc_set(OLEVEL) WHERE Branch_Name = "altA"

To restrict object-level notification to particular objects or types of object change in a public WIP version, data-object-level condition specifications can be sent to the notification server as discussed in Section 8.3.2.2.

This thesis made an assumption (see chapter 3) that change notification can only be requested and occur during client process and notification server lifetimes. Long-term condition specification and notification by means other than inter-process communication (IPC) has been left for future work, but we recognize that some applications which involve versioning (e.g., computer-aided design, software control) are where clients might want to receive notification when they are not active. The DO <action> clause could be used, for example, to request notification via email:

[5] ON UID TO proj1_doc_set() DO email(doc_librarian@cs.sfu)

#### 8.3.2.2 Data-Object-Level Condition Specification

Previously (in section 3.3) we presented a condition specification language that allows DB clients to describe a collection of application class objects that they wish the notification server to monitor for change.

Data-object-level condition specifications can be used to restrict notification to particular objects, or types of object change, in a WIP version. For example, if a DB client wishes to receive notification of changes to a specific document in the proj1_doc_set unit, an object-level condition specification like [6] below can be sent to the notification server after a version-level condition specification like [1] or [2] above has been sent to the server:

[6] ON UID TO Document_Hdr!() WHERE doc_id = "AA-123"

Later, when the notification server receives a data-object-level DB update event message for an object in a Document_Hdr aggregation hierarchy where doc_id = "AA-123", the server will notify the client if either:

. the object is not part of a version unit (i.e., it is not being versioned), or

. the object change was made to a version unit that the client has an active version-level condition specification for (e.g., proj1_doc_set), and the object change was made to the WIP version of the client's current group.

There are four points of interest to note from the above:

. earlier, we restricted our version management model to version units which do not overlap. This means that the notification server does not need to determine if an object is a member of more than the one unit where it was changed (not a trivial problem), in order to figure out if clients interested in other units need to be notified of the change.

. we also restricted our version management model to version units whose data objects do not reference other objects in other version units This means that the notification server does not have to monitor aggregation hierarchies that span version units, or resolve pointers (which may be static or dynamic) to versions of other units. See [CK88] for an implementation which attempts to address these issues in the ORION OODBMS.

. in part because of points one and two above, intra-object conditions in version units are relatively straightforward for the notification server to monitor for change. All the information necessary to determine if a change has occurred for intra-object conditions can be found in a single data-object-level DB update event message.

- inter-object conditions in version units, however, can present complex implementation problems, even when all the objects involved are within one version of a unit. This is because inter-object conditions require the notification server to read data to set up the appropriate events in the monitored data set, and perhaps to maintain partial views. When the server receives an inter-object condition specification, it has to determine which version unit (if any) the objects are in now and then read the WIP in the client's current group to set up its internal data structures. If the client changes groups, or a new WIP version is created, the server must adjust its internal data structures.

### 8.3.3 Acknowledge Condition Specification Messages

Function and format of the acknowledge condition specification message are the same as those given in Section 5.2.3.

### 8.3.4 Cancel Condition Specification Messages

Function and format of the cancel condition specification message are the same as those given in Section 5.2.4.

### 8.3.5 Change Notification Messages

The change notification design given previously in Chapter 5 must be extended in two ways. First, an additional notification message is defined for version-level change. The message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'V' (version-level notification) |
| Timestamp | time message sent |
| Message length | # bytes |
| 'I' \| 'D' \| 'U' | result change (insert,update,delete) |
| Unit name | unique version unit name |
| if 'I' operation: | |
| version ID | branch name, version # |
| if 'D' operation: | |
| delete type | U (unit), W (delete WIP)<br>V (compress versions) |
| version ID | if delete type = 'V' or 'W',<br>branch name, version # |
| if 'U' operation: | |
| update type | CO (checkout), CB (checkout branch),<br>co (cancel checkout),<br>cb (cancel checkout branch),<br>MG (merge), mg (cancel merge),<br>CI (checkin), CA (chg WIP access) |
| version1 ID | branch name, version # |
| version2 ID | if 'MG', second version involved |
| new access | if 'CA', new WIP access for others |

Table 8-4 Version-Level Notification Message Format

Second, the data-object-level notification message format given previously in Table 5-6 must be modified (see Table 8-5 below) to handle the reporting of change to both versioned and non-versioned data. One new field, unit name, has been added to the message so that clients can determine where the object is being versioned.

Note that the WIP version name does not have to be part of the message because each client knows which group it currently belongs to for the specified version unit, and each group can only be updating one WIP version of a unit at a time.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'N' (notification message code) |
| Timestamp | time message sent |
| Message length | # bytes |
| Condition ID | client's unique condition spec. # |
| 'I' \| 'D' \| 'U' | result change (insert, delete, update) |
| CAUSE: | |
| DB update event | 'I' \| 'D' \| 'U' |
| Class name | class of the object that changed |
| Primary key | unique ID of the object that changed |
| Class name | result object's class |
| Unit name | unique version unit name ** |
| After-change attribute values | value of each result object attribute (ordered as listed in the class definition) |
| Attribute name * | if cause event = 'U', full name of the attribute that changed |
| Before value * | if cause event = 'U', value of the attribute that changed |

* field repeated for each attribute modified
** field only used for versioned data; otherwise blank

Table 8-5 Extended Data-Object-Level Notification Message Format

If a client sends a group change message to the server (see Section 8.3.6), the client must wait for the receipt of a group change acknowledge message before assuming change notification messages relate to the new group.

### 8.3.6 Group Change Messages

By default, a client's group affiliation is set for a version unit when it issues a version-level condition specification for the unit. A client may have only one version-level condition specification active in the notification server for a particular version unit because a client can only belong to one group at a time.

To leave one group and join another, a client could cancel the active version-level condition specification and issue a new one, but that would cause unnecessary processing overhead in the notification server. Instead, a group change message is defined which asks the

server to change a client's current group for a particular version unit. The message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'G' (group change message code) |
| Timestamp | commit time |
| Message length | # bytes |
| Unit name | unique version unit name |
| Branch name | unique branch name (new group) |

Table 8-6 Group Change Message Format

The notification server sends a message to clients acknowledging receipt of each group change message. Clients should always check that a 'valid' status code is returned. An 'error' status code will be returned if the unit name or branch name is invalid, or if the client has no active version-level condition specification for the particular version unit (i.e., the group change message cannot be used to initialize a client's current group). The acknowledge message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'G' (group acknowledge message code) |
| Timestamp | commit time |
| Message length | # bytes |
| Unit name | unique version unit name |
| Branch name | unique branch name (new group) |
| Status code | 0 (change accepted); 1 (error) |

Table 8-7 Group Change Acknowledge Message Format

### 8.3.7 Group List Request and Reply Messages

A client can request a list of members in any group at any time. The message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'L' (group list request message code) |
| Timestamp | commit time |
| Message length | # bytes |
| Unit name | unique version unit name |
| Branch name | unique branch name; if blank, default to client's current group |

Table 8-8 Group List Request Message Format

The server sends a group list reply to the client. The message contains the number of clients currently in the group, and a list of socket ids so that clients can choose to initiate a direct dialogue with other members of a group. The message format is given below.

| FIELD | DESCRIPTION |
|---|---|
| Message type | 'R' (group list reply message code) |
| Timestamp | commit time |
| Message length | # bytes |
| Unit name | unique version unit name |
| Branch name | unique branch name (group) |
| Number of clients | total number of clients in the group |
| Client ID * | unique name, or process id |
| Socket # * | socket # of a client in the group |

* field repeated for each client in the group

Table 8-9 Group List Reply Message Format

# CHAPTER 9
# Conclusions and Future Work

## 9.1 CONCLUSIONS

Figure 9-1 below lists our high-level change notification requirements and summarizes how well various notification approaches meet those requirements. We were motivated to develop our change notification server because no embedded approach currently satisfies all our requirements, and we feel that separating the notification mechanism from the DBMS provides more flexibility and extensibility.

| | Embedded in an OODBMS | | | | |
|---|---|---|---|---|---|
| Our Chg Notification Requirements | NOTIFY LOCKS | ACTIVE QUERIES | ACTIVE OODBMS Triggers | ACTIVE OODBMS Rules | NOTIFICATION SERVER |
| Complex condition specification possible | N | Y* (1) | Y* (2) | Y* (1) | **Y** |
| Conditions specified separate from data access | N | N | Y | Y | **Y** |
| Able to indiv. specify kinds of chg (I,U,D) | N | N (3) | Y* (4) | Y* (4) | **Y** |
| Dynamic (run-time) cond. specification (5) | Y | Y | N | maybe | **Y** |
| Tailored to the needs of each active client (6) | Y | Y | N | N | **Y** |
| Chg notification for versioned DB schema | N | N | Y* (7) | N | **Y** |

* has limitations

Figure 9-1 A Comparison of Change Notification Approaches

| Notes for Figure 9-1 |
| --- |
| (1) limited by the capabilities of the embedded (i.e., fixed) query or rule-specification language. Our condition specification language can be tailored to the needs of particular applications by allowing application-specific functions to be used. |
| (2) OODBMS triggers are embedded in class definitions. Although complex conditions can be specified, they are only checked when the object containing the embedded trigger is changed. Notification with triggers is either limited to intra-object conditions, or triggers must be specified in all objects which may be part of an inter-object condition (awkward to verify the correctness of and to maintain, especially in large application DBs). |
| (3) in the current literature, active queries are implemented using the DBMS's passive query language and a set of activate/deactivate functions embedded in the DBMS. Our condition specification language is an active language, and as such, allows clients to specify an interest in particular kinds of change. |
| (4) The format of triggers and rules is, generally: "ON <event> IF <condition> DO <action>". To provide the same range of notification as is possible with an active query, a number of triggers or rules must be specified. Our condition specification language allows clients to specify one or all kinds of change with one statement. |
| (5) by 'dynamic condition specification', we mean the ability to add or activate condition specifications, and delete or deactivate current condition specifications at run-time. |
| (6) Condition specifications and notification must be able to be individualized, since each DB client's interests may be different. Embedded triggers and rules generally affect all currently active clients. Also, clients come and go - embedded triggers and rules don't generally keep track of active clients, so it is difficult to determine who to notify. |
| (7) [CK88]'s version management model is limited to versions of single objects only, and notification is more for internal change propagation than for external DB client monitoring/alerting. Our version management model allows a number of data objects to be grouped into a versionable 'unit', and includes the concept of client groups (which allows the notification server to make condition specification and notification more flexible). |

The major contributions of this thesis are as follows:

(1) examination of object-oriented (as opposed to relational) change notification.

In fact, we have examined relational change notification because its requirements are a subset of the requirements for object-oriented change notification. Relational DBMSs have tables whose data changes through row insert, update, and delete, while OODBMSs have classes whose data changes through instance insert, update, and delete. Whether DB update events, condition specification, and change notification are focused on tables or classes, our basic design remains the same.

However, OODBMSs differ significantly from relational DBMSs in that OODBMSs must support complex inheritance hierarchies and aggregation hierarchies. This work makes an important contribution by examining, in detail, condition specification and change notification for these hierarchies.

(2) description of a flexible and sufficient language for specifying conditions of interest to monitor for change in an OODBMS.

We have combined aspects of both active queries and imperative rules so that DB clients can specify exactly what kinds of changes they are interested in. We have designed the language so that clients can easily specify an entire inheritance or aggregation hierarchy with one condition specification.

We have allowed application-oriented functions to be a part of the language definition so that condition specification can be tailored to the needs of particular applications.

(3) design of a change notification server which is informed of committed data change by DB clients (i.e., the server does not have to intervene between clients and the OODBMS to detect change), and then notifies clients who have registered their interest in the change.

(4) presentation of a version management model, and a detailed examination of how our server design can be extended to handle notification of change to versioned data.

We have added a second set of DB update event, condition specification, and notification messages to the design to address version-level change. The basic design, which addresses data-object-level change, is essentially the same for both versioned and non-versioned DB schema.

We have grouped clients within a version unit by requiring them to focus on one branch of the version graph at a time. Then, the condition specifications a client sends to the notification server do not have to specify which work in progress (WIP) version to monitor for change. The WIP version is given by a client's current group membership, and if that group membership is changed, the server can automatically adjust the focus of a client's active condition specifications.

We have shown that a centralized OODBMS change notification server is possible, that the interface between DB clients and the notification server can be simple but effective (i.e., there are a small number of messages, and message content is relatively simple to generate and interpret), and that condition specification can be dynamic (i.e., clients can issue and cancel condition specifications at any time).

However, our notification server design has several limitations which make it inappropriate for use by some applications. The following is a summary:

- only change made directly by DB clients can be made known to the notification server. Change propagation (e.g., the recalculation of derived values, or cascading delete) is generally considered invisible because it is encapsulated in DB schema class methods.

- when the number or size of object attributes is large, the length of DB update event messages and change notification messages could be a communication and/or performance problem.

- if there are large numbers of DB updates, but few or no conditions being monitored for change, there will be a significant overhead in message passing for which little or no benefit is being derived.

- because the notification server is behind in its processing of DB update event messages, and therefore cannot read the DB for information when evaluating some inter-object condition specifications, the server must keep some partial views that are optional in other approaches to change notification (i.e., embedding notification in the DBMS).

- the change notification server is "stateful"; that is, if the server crashes (or is shut down while in use by clients) it loses all knowledge of active clients and their condition specifications. This thesis does not address how clients and a restarted server might re-establish communication - we recognize that server failure recovery is not a trivial problem, and have left it to future work.

In addition, the notification server can be overloaded by large numbers of incoming and outgoing messages, and/or having to parse and evaluate a large number of complex condition specifications. These performance limits are not unique to our approach - they are expected of any central-server design, and of any centralized approach to change notification (i.e., embedded active queries or rules) that must evaluate complex condition specifications. Like query processing, our approach pays an additional performance penalty by allowing dynamic condition specification - the server must parse condition specifications and map them into the monitored event set as they arrive, rather than being able to preprocess them.

If an implementation of our change notification server existed, it would take application programmer(s) some time and effort to use it. An application DB schema knowledge base (KB), and an application-oriented function KB (if any functions were to be used - an examination of potential condition specifications would determine this) would have to be created. Class definitions and data access routines for any application data that the notification server might need to read would have to be linked into the server. If any aggregation hierarchies were to be monitored, they would have to be studied to determine what DB update events should be reported, and what notification should be expected from condition specifications (as discussed in Section 4.2.2). A library of server communication routines would likely be provided for clients to call, but application programmer(s) would have to provide software to generate the content of DB update event messages for each application class, and to parse the content of change notification messages for each application class.

Despite the effort required to use the change notification server, it is not excessive, nor is it difficult to understand what has to be done and why. The effort would be worth it if polling is not a desirable option for an application's DB clients (e.g., when the amount of

data being monitored is large, or when timely response is required). Therefore, we can conclude that our change notification server would be a valuable component for many applications.

## 9.2 FUTURE WORK

In the future, it would be desirable to do the following with our current design:

- implement the notification server, and prototype the two example applications
- study the requirements of the Function KB, and determine how application-oriented functions in condition specifications could be implemented
- analyze notification server performance
- apply change notification to a broader set of applications, including an application with versioned data
- study the impact on the notification server design and implementation when it is applied to other OODBMS and to RDBMS.

In the future, it would also be desirable to study extensions to the notification server design, including (in no particular order):

- broadening the kind of change that the server can notify clients of. Currently, the server only sends notification of persistent data change. However, we discovered at least two cases (see Sections 4.3.4.2 and 7.1) where clients would benefit if they could receive notification when another client terminated unexpectedly, and although we defined a group list request message for versioning applications, clients do not get notification of change to the group. We could modify the server design to send

notification to clients when others establish communication with the server, terminate, join or leave a configuration group.

- allowing long-term condition specification (i.e., storing condition specifications in the database) and allowing notification when clients are not active (e.g., via email).

- handling applications whose data is distributed in two or more databases

- handling correlated data change; that is, collecting all the DB update event messages for one transaction, and assessing the net effect of the changes before sending out notification messages.

- allowing uncommitted DB update event messages (via methods embedded in DB schema class definitions) so that the notification server can be informed of change propagation. Note that change notification would still be for committed change.

- investigating the need for, and requirements of, uncommitted change notification

- adding an option to the condition specification language so that clients can indicate whether or not they wish to be informed of their own changes. Currently, they are always informed of their own changes if they have an active condition specification that requests notification of the changes.

- investigate the design of a distributed change notification server

# APPENDIX A
# ObjectStore Version Management

This appendix is included in order to give the reader some idea of how ObjectStore's implementation (Release 1.2) of version management compares with the model of version management presented in Chapter 8.

There are two significant differences, namely ***configurations*** and ***workspaces***, for which an overview is provided in the sections which follow. ObjectStore also allows inter-object references (static or dynamic) between configurations, has two more version-level update events ('new_version', and 'freeze'), and appears to allow single version delete under all but one circumstance ([OD91b] indicates that all but the most recent read-only version on the initial branch may be deleted).

We do not include a detailed discussion in this appendix of how these differences affect our change notification design, but we have possible solutions to all but the inter-object references between versions of different configurations (which we have left for future work). Therefore, we can say that our change notification server can be made to support much of ObjectStore's version management model, but at the cost of making it much more complex and ObjectStore-specific.

## A.1 CONFIGURATIONS

In ObjectStore, a configuration is the unit of versioning, but its definition is different from what we called a 'version unit' in Chapter 8 because ObjectStore allows subconfigurations.

For example, let's use the document set example once more - in ObjectStore, proj1_doc_set is defined as a configuration, and each document is defined as a subconfiguration. When a version of a configuration is checked out or checked in, ObjectStore's default behaviour is to propagate the action to all its subconfigurations, and then their subconfigurations, and so on. This recursive behaviour can be overridden when necessary to create a configuration which consists of mutually consistent versions of a set of documents as the following scenario illustrates:

Version-1 of proj1_doc_set is created containing version-1 of document A and version-1 of document B. Three versions of document A (V2 thru V4) are subsequently created (via checkout/checkin of just the Document_Hdr subconfiguration) before its text is considered complete enough to be included in version-2 of proj1_doc_set.

The second version of proj1_doc_set is created by checking it out with recursive behaviour turned off so that version-5 of document A and version-2 of document B are NOT created. When version-2 of proj1_doc_set is checked in (again with recursive behaviour turned off), ObjectStore has to resolve version-2's subconfiguration references to some objects, and it chooses the most recent versions of document A (V4) and document B(V1).

Figure A-1 shows what the version graph would look like after version-2 of proj1_doc_set is checked in. Note that V2 and V3 of Document A are not referenced by any version of proj1_doc_set, but they can be accessed through the use of ObjectStore's predecessor (i.e., read_prev) and successor (i.e., read_next) functions on document A.

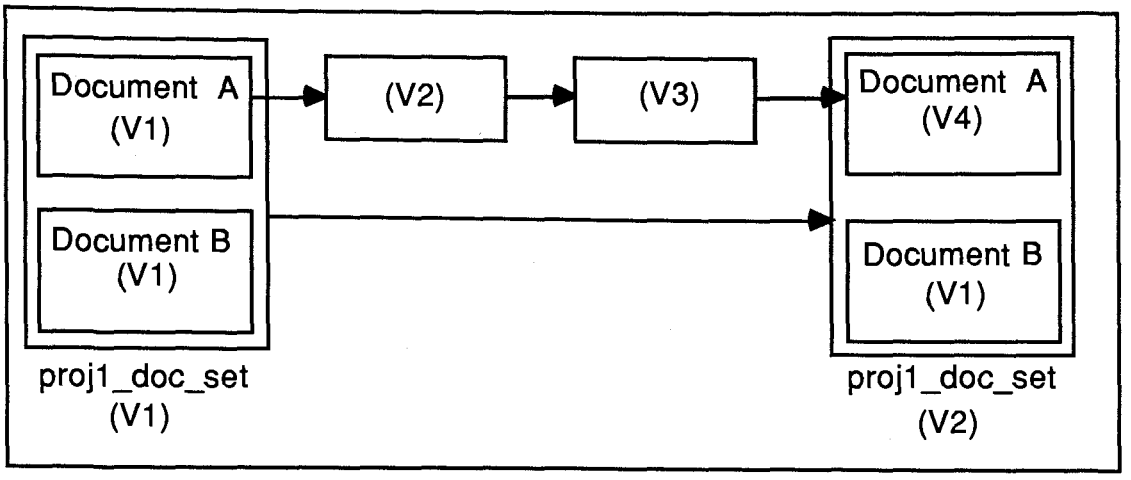



Figure A-1 Subconfiguration Example's Version Graph

## A.2 WORKSPACES

In our version management model, the rules of configuration/version sharing are simple. All read-only versions of all configurations are accessible by all clients. The WIP versions of configurations are controlled (owned) by the clients that checked them out. Other clients are unable to access a WIP version unless the owner makes the WIP version public, and the degree of sharing granted by the owner may be read-only, read-write, or read-write-delete-checkin.

In ObjectStore, sharing is controlled by workspaces; configurations/versions are owned by a workspace (WS), not one or all DB clients. A client's current workspace controls what configurations, and what versions of those configurations, are visible to the client. If a WIP version is in a client's current workspace, the client has read-write-delete-checkin access.

Workspaces form a hierarchy that can have any number of levels. Configurations/versions in a parent workspace are visible from its descendants, but configurations/versions in a

child workspace are not visible from its ancestors. Thus, nodes higher in the hierarchy have higher visibility (i.e., are more public).

A DB client checks out a version from a parent workspace into a child workspace, makes updates to the WIP version over some period of time, then checks the WIP version back in to the parent. Other clients whose current workspace is the parent will not be able to access the WIP version until it is checked back in. Note that if a client knows the child WS name, there is nothing in ObjectStore to prevent a client from changing its current workspace from the parent to the child in order to gain access to a WIP version.

Different applications will define different workspace hierarchies, and will store configurations in different nodes of the hierarchy. In ObjectStore, clients are grouped based on their current workspace, so the WS hierarchy design will depend on how clients in the application will share versioned data. For example,

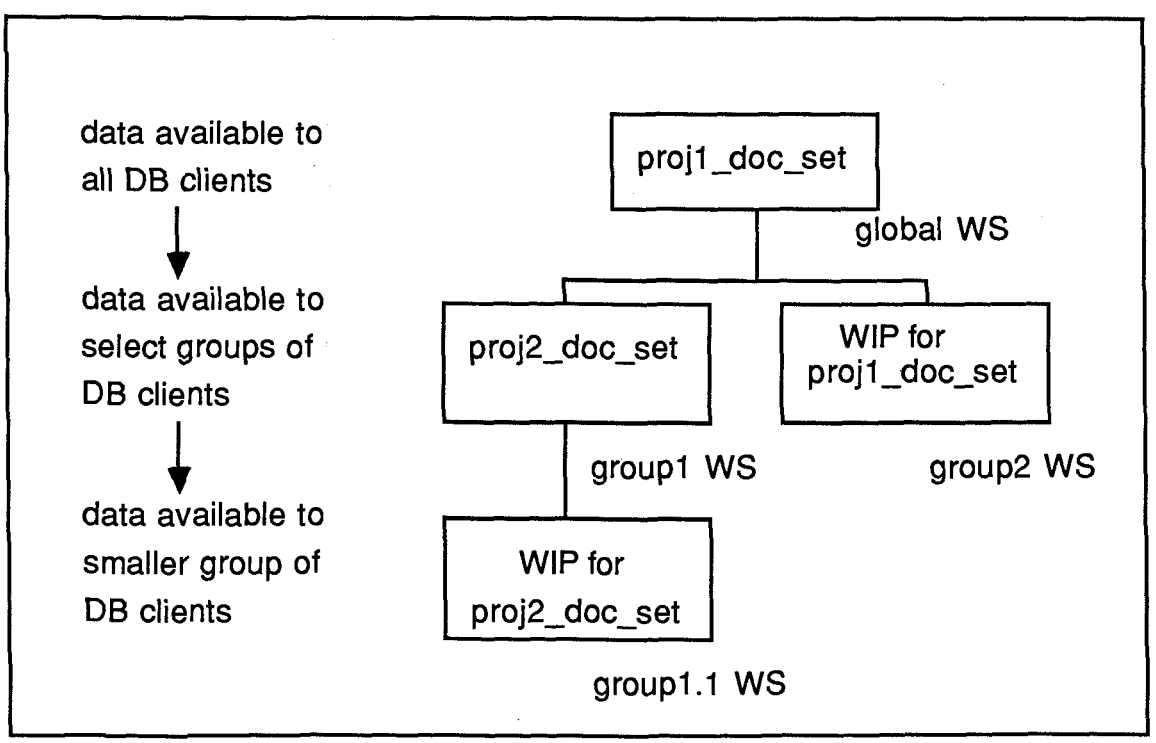



Figure A-2 A Workspace Hierarchy Example

In Figure A-2, the proj2_doc_set configuration is only visible to those clients whose current workspace is *group1* or *group1.1*. The versions of the proj1_doc_set configuration in the *global workspace* are visible to all clients, no matter what their current workspace is.

# APPENDIX B
# Server Data Structure Design

In this appendix, we describe two of the notification server's data structures, the application schema KB and the monitored event set, in more detail. The information they must contain is given, but their physical structure when implemented may be different from what we present here.

## B.1 THE APPLICATION SCHEMA KNOWLEDGE BASE

The following classes define the application schema knowledge base structure:

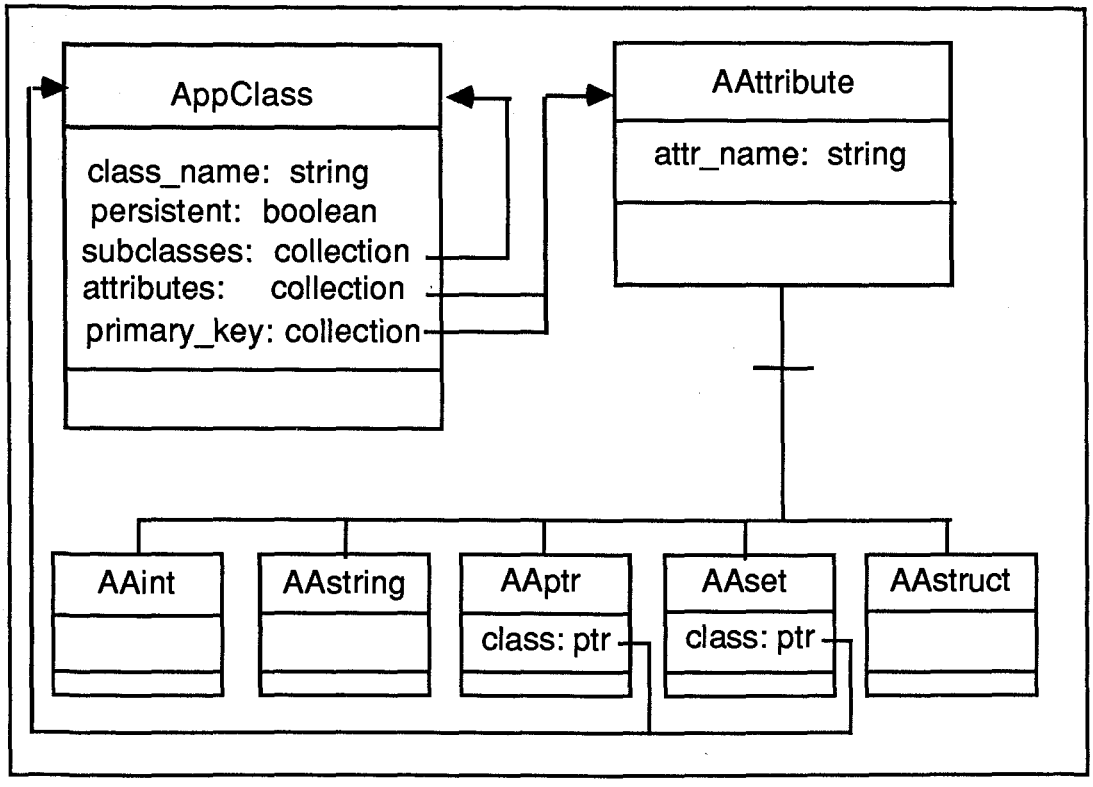



Figure B-1 Application Schema Knowledge Base Structure

There is one AppClass instance for each application schema class. AppClass attributes indicate if persistent instances of the class are possible (i.e., is the class an abstract base class or not), document class hierarchy connections, list the class attributes, and identify the primary key (if a composite key, then the primary_key collection must be ordered).

There is one AAttribute subclass (AAint, AAstring, etc.) instance for each class attribute, and each common structured type (e.g., date, time). The number of AAttribute subclasses depends on the number of attribute types expected to be used by application data. Figure B-1 does not show all the subclasses that are possible. The attribute information is used for validation, and to document aggregation hierarchies (i.e., AAptr and AAset attributes are inter-object connectors).

The definition of each AAttribute subclass must provide the necessary information for the server to do the desired level of condition specification validation. For example, given a condition specification like the following:

[..] ON IUD TO Station() WHERE capacity > 9

the notification server could reject this if it knew that the valid range for the capacity attribute was from 1 to 5 (i.e., don't monitor for a condition that will never occur). We expect that an implementation of the notification server will minimize validation since many condition specifications will be fixed in application software. If condition specifications may be entered dynamically by a user, validation is more critical.

The following is an illustration of part of the schema knowledge base for the railway network application (see Section 4.2.2 for the DB schema diagrams):

| AppClass | | | AAtribute | |
|---|---|---|---|---|
| Train | F | (Passenger_Train, Freight_Train) | train_id | int (PRIMARY KEY) |
| | | | status | int |
| | | | speed_lim | int |
| | | | cur_speed | int |
| | | | locn | struct (stn_id, km_to_stn) |
| | | | update_counter | int |
| Passenger_Train | T | () | num_passengers | int |
| Freight_Train | T | () | hazardous_cargo | int |
| Train_Schedule | T | () | train_id | int (PRIMARY KEY) |
| | | | cur_seg | int |
| | | | segs | set of Tsched_Segment |
| Tsched_Segment | T | () | parent | ptr to Train_Schedule |
| | | | seg_id | int |
| | | | from | int |
| | | | to | int |
| | | | depart | struct (date, time) |
| | | | arrive | struct (date, time) |
| N/A | | | date | struct (yr, mth, day) |
| | | | time | struct (hr, min, sec) |
| | | | yr | int |
| | | | mth | int |
| | | | day | int |
| | | | hr | int |
| | | | min | int |
| | | | sec | int |

## B.2 THE MONITORED EVENT SET

The following classes define the monitored event set data structure:

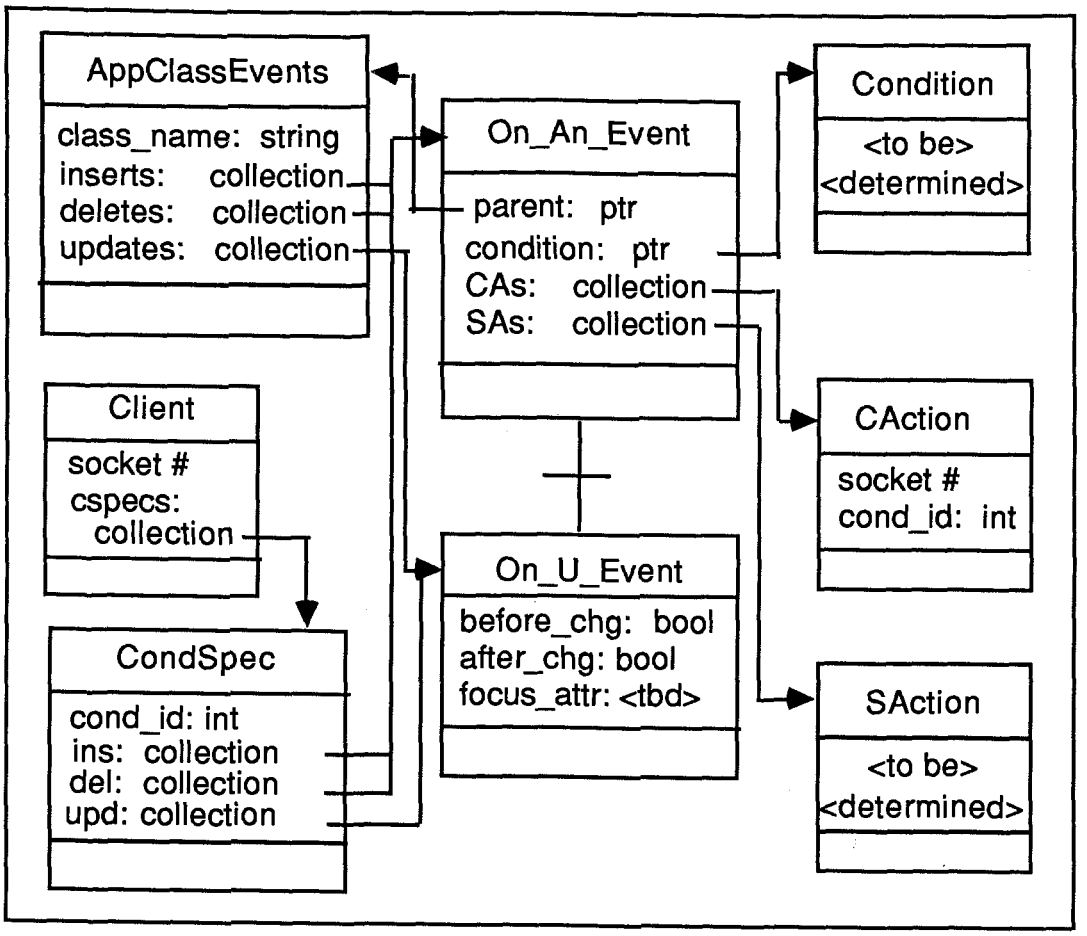



Figure B-2 Monitored Event Set Data Structure

There is one AppClassEvents instance for each application schema class being monitored for change by the notification server. The application schema classes are those with persistent instances; abstract classes are not monitored. Each application class may have 0..n insert, delete, and update event conditions being monitored.

On_An_Event, and therefore On_U_Event, instances have four (4) attributes: a pointer to the parent AppClassEvents object (for finding the parent when deleting condition specifications), a pointer to a Condition object, a collection of pointers to client action

objects, and a collection of pointers to server action objects. The last two attributes are collections because more than one client may need to be notified and more than one action may need to be taken by the server.

On_U_Event instances have additional attributes as described in Section 6.1.2.1.

A Condition instance contains the contents of a condition specification's WHERE clause. A Condition can be shared by any number of On_An_Event and On_U_Event instances. The definition of the Condition class is designated 'to be determined' because we have left the implementation of predicate evaluation for future work.

CAction instances provide the information necessary to send a notification message to client process(es) when a change occurs. If we assume only notification to active DB clients, CAction attributes are the client's socket number (how to send the message) and the client's condition specification ID (what condition spec. the change refers to). If notification could take another form (e.g., email), the CAction definition would have to be extended.

SAction instances provide the information necessary for the server to update its partial views and check complex results when a change occurs. The SAction definition is designated 'to be determined' because we have left the implementation of complex condition specifications for future work.

There is one Client instance for each active DB client process connected to the server. The cond_specs collection is empty until a client sends the notification server its first condition specification. There is one CondSpec instance for each active condition specification. CondSpec instances allow the server to cancel condition specifications without having to search the AppClassEvents aggregation hierarchy for all CActions associated with a particular client or cond_id.

# References


[BC79] O. Peter Buneman and Eric K. Clemons, "Efficiently Monitoring Relational Databases", *ACM Trans. on Database Systems*, Vol.4, No.3, Sept. 1979, pp. 368-382

[BeM91] Catriel Beeri and Tova Milo, "A Model for Active Object Oriented Database", *Proceedings of the 17th International VLDB Conference*, Barcelona, 1991, pp. 337-349

[BM91] Elisa Bertino and Lorenzo Martino, "Object-Oriented Database Management Systems: Concepts and Issues", *Computer*, April 1991, pp. 33-47

[CK88] Hong-Tai Chou and Won Kim, "Versions and Change Notification in an Object-Oriented Database System", *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988, pp. 275-281

[CN90] Sharma Chakravarthy and Susan Nesson, "Making an Object-Oriented DBMS Active: Design, Implementation, and Evaluation of a Prototype", *Proceedings of the International Conference on Extending Database Technology*, Italy, 1990, pp. 393-406

[CS92] Raymond G.A. Cote and Ben Smith, "Tapping into Sockets", *Byte*, March 1992, pp. 261-266

[Date90] C.J. Date, "An Introduction to Database Systems, Volume 1 (5th ed.)", Addison-Wesley, 1990

[Day88] Umeshwar Dayal, "Active Database Management Systems", *Proceedings of the 3rd International Conference on Data & Knowledge Bases*, Jerusalem, June 1988, pp. 150-169

[Day91] Umeshwar Dayal, "Tutorial 8: Active Database Management Systems", notes distributed at *IEEE 7th International Conference on Data Engineering*, 1991, pp. 1-62

[DMFV90] David J. DeWitt, David Mater, Philippe Futtersack, and Fernando Velez, "A study of three alternative workstation-server architectures for object-oriented database systems", *Proceedings of the 16th International VLDB Conference*, Brisbane, 1990, pp. 107-121

[DPG91] Oscar Diaz, Norman Paton, and Peter Gray, "Rule Management in Object Oriented Databases: A Uniform Approach", *Proceedings of the 17th International VLDB Conference*, Barcelona, 1991, pp. 317-326

[FVFH90] J. Foley, A. Van Dam, S. Feiner, and J. Hughes, "Computer Graphics (2nd ed.)", Addison-Wesley, 1990

[GJ91] N. Gehani and H.V. Jagadish, "Ode as an Active Database: Constraints and Triggers", *Proceedings of the 17th International VLDB Conference*, Barcelona, 1991, pp. 327-336

[Hor86] R. Nigel Horspool, "C Programming in the Berkeley UNIX Environment", 1986

[HZ87] Mark F. Hornick and Stanley B. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database", in *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990, pp. 273-285

[Kim89] Won Kim, "An Approach to a Total Solution to Long-Duration Transactions", MCC Technical Report ACT-OODS-223-89

[KP88] Glenn E. Krasner and Stephen T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *JOOP*, Aug/Sep 1988, pp. 26-49

[LC91] W.S. Luk and Amelia Choi, "Dynamic Spatial Query Language: A Customized Query Language for Object-Oriented Database Systems", *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC)*, 1991

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb, "The ObjectStore Database System", *Comm. of the ACM*, Vol. 34, No. 10, October 1991, pp. 50-63

[Mey91] Scott Meyers, "Difficulties in Integrating Multiview Development Systems", *IEEE Software*, January 1991, pp. 49-57

[OD91a] Object Design Inc., "ObjectStore User Guide", Release 1.1, March 1991 (DN110SUN-DEV)

[OD91b] Object Design Inc., "ObjectStore Reference Manual", Release 1.1, March 1991, (DN110SUN-DEV)

[Rei90] Steven P. Reiss, "Connecting Tools Using Message Passing in the Field Environment", *IEEE Software*, July 1990, pp. 57-66

[Ris89] Tore Risch, "Monitoring Database Objects", *Proceedings of the 15th International VLDB Conference*, Amsterdam, 1989, pp. 445-453

[Shan90] Yen-Ping Shan, "MoDE: A UIMS for Smalltalk", *ECOOP/OOPSLA '90 Proceedings*, Ottawa, October 1990, pp. 258-268

[SPAM91] Ulf Schreier, Hamid Parahesh, Rakesh Agrawal, and C. Mohan, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS", *Proceedings of the 17th International VLDB Conference*, Barcelona, 1991, pp. 469-478

[Sun90] Sun Microsystems, "Sunview Programmer's Guide", Part# 800-1783-11, Revision A, March 27, 1990

[Wiss90] P. Wisskirchen, "Object-Oriented Graphics: From GKS and PHIGS to Object-Oriented Systems", Springer-Verlag, 1990

[WN90] Kevin Wilkinson and Marie-Anne Neimat, "Maintaining Consistency of Client-Cached Data", *Proceedings of the 16th International VLDB Conference*, Brisbane, 1990, pp. 122-133

[ZM90] Stanley B. Zdonik and David Maier (eds.), "Readings in Object-Oriented Database Systems", Morgan Kaufmann, 1990, p. 25