Notification in the THOR Database System

by

Paul H. Kim

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science at the

Massachusetts Institute of Technology

May 24, 2002

Author ____

Department of Electrical Engineering and Computer Science
May 17, 2002

Certified by ____

Dorothy Curtis
Thesis Supervisor

Accepted by ____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Notification in the THOR Database System

by

Paul H. Kim

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science at the
Massachusetts Institute of Technology
May 24, 2002

Abstract

There is an increasing amount of data in the world that is distributed over many machines. Some of this data may be relatively static. However, certain types of data may be changing frequently. Client applications may be interested in changes that are made to a small subset of the data. Notification of such changes allows a client to be made aware of changes that are made by another client application in a timely fashion. In this paper, we explore the implementation of a notification mechanism in the unique object−oriented environment of the THOR distributed database system.

ACKNOWLEDGEMENTS

I would like to express my heartfelt thanks to my advisor, Dorothy Curtis, for her invaluable contributions that went well above and beyond that of a thesis advisor. Her thoughtful support and guidance throughout the duration of this project truly helped make this the most satisfying learning experience of my academic career.

I would also like to thank Sidney Chang for so selflessly giving of her time to show me the in’s and out’s of THOR.

Table of Contents

1. Introduction

1.1 Motivation

With the increasingly distributed nature of applications, clients can now assume a global context for both distributing and receiving data. With the large amount of data that is available to clients in distributed systems, it is possible that the client may be interested in changes made only to a certain subset of the data.

A potential use for notification is in a temperature monitoring application, where temperatures are stored in a database. The monitoring application is interested in changes made to the temperature values, so that it can take the appropriate action once the temperature reaches a certain threshold. If the database is storing a lot of dynamic information from other sensors, the data that the monitoring application is interested in represents only a small subset and a selective notification mechanism would be useful to such an application.

Notification has been implemented in various systems. This thesis explores the issues in adding notification to an object−oriented database system that uses an optimistic concurrency scheme: THOR.

1.2 Solutions: Polling Vs Notification

One possible solution is to have the client poll the system to check whether the value of time−to−arrival has changed. Although the application may be able to make general estimations on when the value will be modified, it has no means of precisely determining the exact instant the value is changed. Repeatedly making calls over the network connection to inquire about the value could potentially tie up the network. Such network calls become even more costly in a scenario where the connection must repeatedly be established by time−intensive dial−up connections. Alternatively, having the system notify the waiting client application upon change of time−to−arrival requires only one network call at the precise time that the value is actually modified. Hence, notification provides two benefits over polling: preservation of network bandwidth as well as providing a new value at time closer to that at which the value was changed.

Additionally, notification can be used in conduction with polling to improve the latency at which data is refreshed. If an application periodically polls for updated values of a piece of data it is interested in, the data at the time immediately before a polling call is made is guaranteed to be no fresher than the periodicity of the polling. If notification is integrated with the polling, however, we can see improvement on this latency. The data will be guaranteed to be at least as fresh as the polling would provide, and in all likelihood, more fresh than polling.

1.3 Notification within the THOR Database System

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

This paper will discuss the implementation of a notification mechanism in the THOR database system that notifies a client application upon a change to a given piece of data. Furthermore, the system will address the issue of providing the client with an updated copy of the modified data. We seek to provide the notification mechanism within THOR in a manner that maintains consistency, promotes the rapid update of stale data, and efficiently makes use of the limited network bandwidth and CPU processing resources in the system.

By notifying the client application of object modification, THOR will able to implement a wait−for−change functionality. Essentially, the client applications will be able to "expect" another application to change the value of one of its object. By providing a copy of the new object, the application can be assured that the its transactions are being performed on valid objects. The overhead cost of polling the server to check if the object has been modified will no longer have to be undertaken by the client, nor will the network be burdened by a potentially heavy load of polling calls.

1.4 Thesis Outline

The remainder of the thesis will be as follows: Section 2 will provide an overview of the THOR architecture as well as introduce the notion of data consistency. Section 3 will discuss the details and implementation details of the notification mechanism within THOR. Section 4 will explore related work, and Section 5 will conclude with possible future projects that may benefit from notification.

2. Background

This section provides insight into the issues and concepts involved with notification. Furthermore, we examine the definition of notification itself, and identify the applications that use notification as well as the various types of notifications that are possible. Finally, a summary of the THOR architecture is provided with a look at notification in the context of THOR.

2.1 Consistency and Invalidation

There are essentially two types of data consistency enforcement among objects of a distributed system: absolute and relative. Absolute consistency requires that the copies of the objects held by the users (the applications in our case) are consistent with the centralized copies stored at the database at all times. Relative consistency, on the other hand, demands that the each of the members of the set of objects contained by one application are consistent with one another at a specific time t. In other words, relative consistency can be maintained within an application as long as the data represent the values that were shared together at one point in time.

An application requiring relative consistency does not place emphasis on the freshness of its data. For example, an application that attempts to correlate the time of day to the traffic on a certain road require only that the values for the time of day and the number of cars on the road be taken at the same time. It does not need the two values to be up−to−date. The fact that one value may have changed does not jeopardize the correctness of the system. Increasingly more applications, however, require consistency throughout the entire system. An example of such an application would be a bank account manager. If two parties were simultaneously accessing an account, the system should ensure that balances shown to each party are in synch with each other. Thus, absolute consistency is required in applications that demand the freshest data.

Invalidation is a means to preserve consistency within a distributed system. Invalidation refers to the process of rendering a subset of an applications object set as being stale, and therefore unusable. Invalidation is used to maintain the correctness and data consistency of the system. Invalidations often result in an applications transaction being aborted, because allowing a transaction to proceed involving stale data could possibly introduce data inconsistencies in the system. For example, if there are two users that share the same bank account, and one user deducts a certain amount from the account, the other user's account is now invalid, if left unchanged. Any transaction he initiates that involves the bank account will not be valid, because someone has changed its value. We will see how invalidation is linked to notification in later sections of this paper.

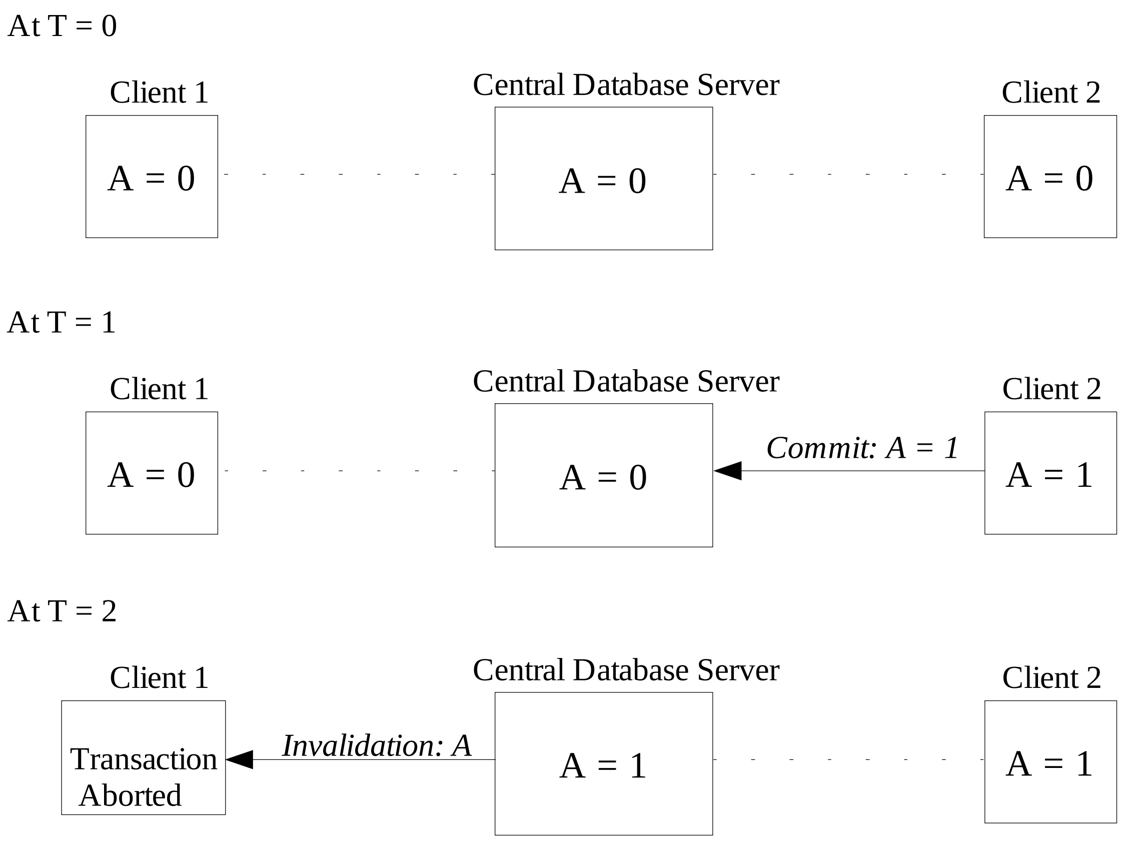


Figure 1: Invalidation preserving data consistency

Figure 1 shows a simple example of the invalidation mechanism involving a central data server and two clients that share a common piece of data A. At time T = 0, ‘both clients have locally cached copies of A. The values of A are consistent with the central server at this point. Client 2 modifies A during a transaction and commits the changes to the central data server. The server updates its own copy of A. Note that this update introduces inconsistency among the shared copies of A. To remedy this, the OR sends an invalidation message to the party which has stale data, Client 1. Client 1 receives the invalidation message for A, and thus aborts its current transaction involving A. Data consistency is thus restored to the system.

Notification in this paper will seek to provide a system with absolute consistency. Applications increasingly operate in a global context, and thus data consistency limited to the application’s own local realm is often not sufficient. Relative consistency greatly limits the capability of applications to have global impact. Notification aims to provide a globally consistent environment, in which applications are locally aware of the global transactions of other applications.

2.2 Notification

The discussion of notification for the remainder of this paper will be in the context of a multi−user, multi−application distributed environment. We will assume that all applications run on top of a shared database, and are connected to the database via a network connection. Furthermore, we will assume that applications work on locally cached copies of data separate from the database. Notification refers to the update of applications of changes made to a given piece of data by another application. It allows for an application to not only be aware of a stale object, but to refresh it as well. Figure 2 displays the semantics of notification. Note how it differs from invalidation and maintains the validity of the transaction.

Notification is an application specific entity. Each application running on the database may be interested in notification for different subsets of the centralized data objects. The central database thus needs state to keep track of which applications are to be notified of which objects. It is possible given any class of applications that there may be commonalities in notifications across applications. In other words, bundling may occur where multiple applications are interested in the same group of objects. In such cases, a multi−cast notification may be appropriate. For the purposes of this paper, however, notifications will be dealt with assuming a one−message−per−application environment.

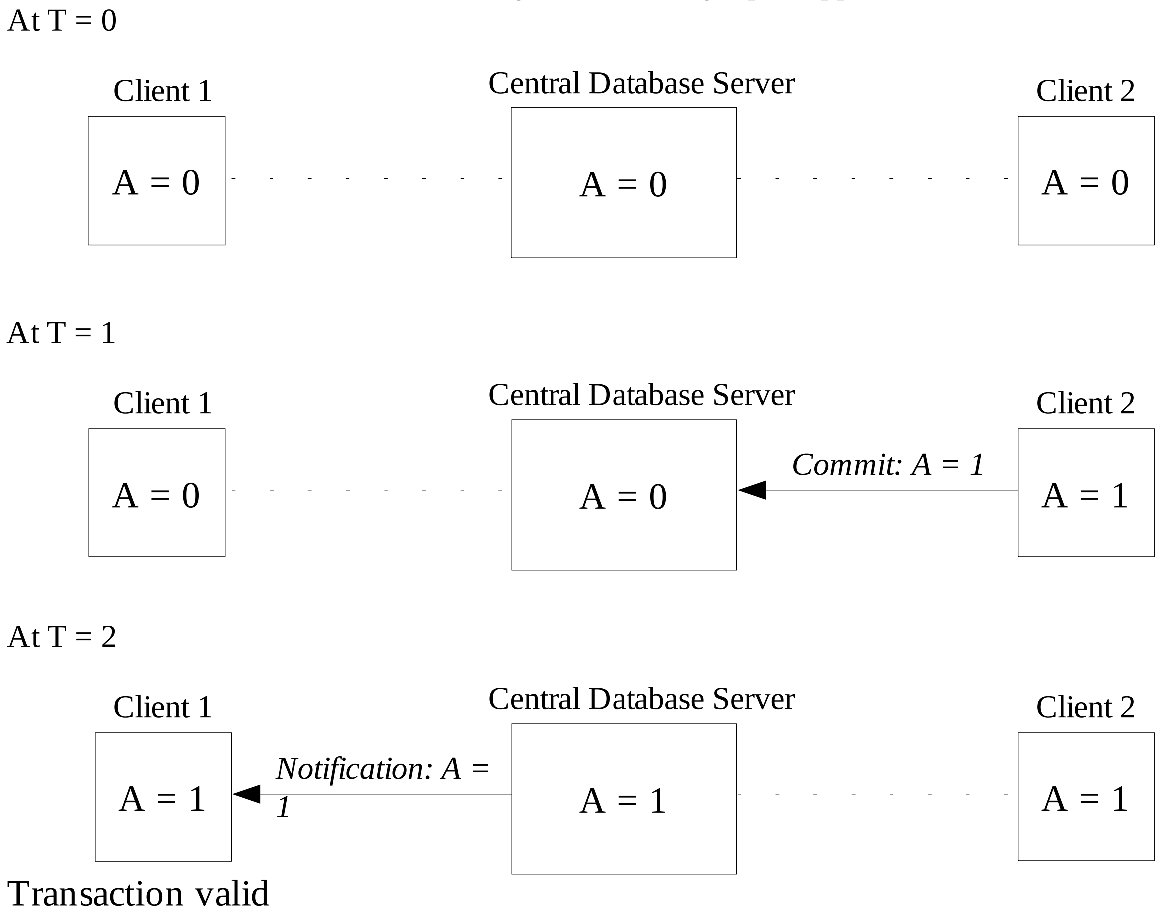


Figure 2: Notification

The class of applications that notification is provided for has great impact on the design and implementation of the mechanism. Clearly the greater flexibility the system provides (i.e, allowing notification for a wider class of applications) the more complexity is introduced into the system. Increased complexity in this case translates into a more complicated scheme to deal with concurrency issues that maintain consistent data. Two distinct types of applications exist for notification: waiting and non−waiting applications. Each has a varying impact on the complexity in maintaining data consistency.

2.2.1 Waiting Applications

Waiting applications refer to those applications that, while waiting to be notified of a change, do not require any other access to or modifications of data. The system therefore does not have to provide access to data during the period of time in which the application is waiting for a notification. This is the simplest class of applications to consider, as no additional concurrency issues are introduced into the system by adding the notification mechanism. The example of the flight arrival notification system mentioned in the Section 1 is exactly such an application. The application does not require access to any pieces of data other than the time−to−arrival, for which it awaits modification.

The impact that implementing notification has on the complexity of such a system is minimal, because there are no potential concurrency issues that need to be addressed. If the application is not reading or writing any other piece of data in the system, then there is no danger of operating on stale or invalid data. By waiting for notification, the application ensures that the one piece of data it is interested in (in this case, the time−to−arrival value) will be current upon receipt of notification. Notifications are thus essential to the continuity of the application, as it will proceed only upon receiving the update.

2.2.2 Non−waiting Applications

Non−waiting applications encompass those applications that actively read and write data, even while potentially waiting for a notification of an update to a member of its data set. Such applications proceed without waiting for notifications, and thus do not require them for the continuity of the application.

In non_waiting applications, the benefit of notification is that applications can better ensure that the data being used in a transaction is fresh, and thus the result of the transaction is more likely to be valid. The number of aborts due to the transaction being invalidated is thus reduced. An example of a non−waiting application that uses notification is a calendar application that can be accessed by two different parties, for example, a doctor and his secretary. In the event where a secretary commits an appointment to a day on the doctor’s schedule, that day becomes invalid to the doctors application. Without notification, the doctor may not know about the change until he tries to add another event to that day, at which point, the transaction will abort, and he can request the new copy of the day.

In contrast, if the system were to notify the doctor of the secretary−entered appointment before the doctor tried to modify the date, the doctor would be aware of the change and be able to act accordingly on a valid piece of data. In such cases, notification can potentially eliminate the need for unnecessary transaction aborts by providing updated and valid copies of data. If the user is notified of an update before he/she attempts to modify data, time will not be wasted on performing invalid transactions.

The aforementioned example illustrates the beneficial case where notification is received in time to prevent invalid transactions. We now consider the case where the notification is not received in time. Using the previous example, if the doctor’s application were to first modify the date by adding an appointment, and then receive the notification that the date had already been changed before it committed to the database, the transaction would be invalid. If the doctor’s application ignored the notification and attempted to commit at this point, the semantical constraints of the database that preserve correctness would not allow the transaction and force the application to abort. The notification could still be of use, as it could allow the application to undo its uncommitted transaction and re−attempt it with the updated value. This would thus save the time required for a commit to be attempted and subsequently aborted.

Semantically, each client of the database keeps track of objects that have been read and objects that have been written to, during the course of a non−committed transaction. These will be referred to as the Read−Object Set (ROS) and Modified−Object Set (MOS) for the remainder of this paper. The members of the ROS and MOS are purged every time a transaction is successfully committed to the database. Upon receiving a notification message for a modified object, the client must check if either its ROS or MOS contain the modified object. If either of them do, the application can infer that its transaction involved the modified object, and is thus invalid. In such cases, the application should abort.

2.2.3 Notification Propagation

There are three levels of the client to which the notification message may propagate: the application cache, the application, or the user. The application and its usage determine how far notifications must propagate for correct behavior. Generally, the more frequent the notifications are expected, the less propagation is necessary. A message that propagates to only the application cache results in the cache updating the applications data object without the application or user having any knowledge of a change. Applications in which the rate of data change is extremely fast, such as real−time stock quoting applications can assume that data is constantly changing, thus do not need to be notified every time a member of its cache is updated. Indeed, application notification would be burdensome and not particularly useful. Applications that explicitly wait for a change in a specific piece of data, however, do require to be notified when the data is modified. Such applications require application−level notification. For example, an application that waits for a temperature to exceed a certain threshold before acting, must know when the notification is received to be able to proceed. Blindly updating the application's data is not sufficient. Propagating the notification message all the way to the user consists of the application somehow conveying an update through its interface to the user. This is needed when changes may impact the users interaction with his/her application. An application that would need to notify its user is the aforementioned calendar application. If a doctor's secretary added a meeting to his calendar, the doctor's calendar application should have some mechanism of notifying him/her of this change.

2.2.4 Notification and Polling

While notification may exist as an alternative to polling in situations where conservation of network bandwidth is desirable, situations in which this is not an issue could couple notification with polling to improve the likelihood of obtaining a fresher value of a particular piece of data. For example, if an application were to poll a database for an updated value every 20 seconds, the value of the data could be as old as 20 seconds by the time the application had access to the value (disregarding the network latency). If polling were then augmented by a notification mechanism, we could ensure the following: in the case that the notification is successfully transferred to the application via the network, the application would have access to the new updated value faster than through simple polling. If the notification is not transmitted correctly, then the underlying polling will ensure that the value is received in a time no slower than simple polling. Clearly, the average case allows for the application to receive an updated value in a shorter amount of time. In instances where the freshness of a value is highly critical, augmenting polling with notification clearly decreases the average latency through which data consistency is established.

2.3 Types of Notification

There are two general types of notification techniques in a distributed environment: message−based and flag−based. In message−based notification, the system notifies its clients of modified objects. Message−based notification is further delineated into immediate or deferred. Immediate message−based notification requires that the clients are notified immediately after the changes to an object are committed, whereas deferred notification allows for the notification to take place at a time specified by the user. In flag−based notification, the system simply updates the data structures that it maintains, so that users will have knowledge of the changes only when they specifically access the object. This paper will focus on an implementation of immediate notification, as the semantics of the flag−based approach are functionally equivalent to a systems invalidation mechanism, as described in Section 2.1.

The deferred method could be implemented by having the application check periodically for any receipt of notification. This check could be an explicit call by the user or be completed unbeknownst to the user, depending on the applications context. Clearly, there will exist some lag time between the notification message being sent, and the application picking up the message. Alternatively, immediate notification forces the application to receive the notification as soon as it is received from the network. This could be accomplished by having the application assign a single processor thread whose sole purpose is to wait for notifications, thus allowing the application to be aware immediately upon receiving the message.

2.4 Reliability & Correctness

Implementing notification decreases the latency of an updated value propagating throughout the system, but also introduces complexity issues that must be dealt with to ensure reliability and data consistency. The reliability of the notification message transmission and handling protocol necessitate a definition of correct behavior that will maintain the system's data consistency. This section examines those issues.

2.4.1 Reliability of Message Transmission

The system sends notification messages to an application via a network connection that is assumed to be inherently unreliable. In other words, there is no guarantee that a single notification message sent to an application will be received, due to unplanned occurrences such as network partitions. The issue of the reliability of notification within a system is thus brought into question.

One possible solution is to implement an acknowledgment response between the client and central server. In other words, the client, upon receiving a notification, sends an acknowledgment back to the server. But once again, there is no guarantee that the acknowledgment will reach the system. Alternatively, we know that the database system is aware when network partitions occur, and is therefore able to have knowledge of which notification messages may not have reached the applications. It will then re−send the notification messages. On the client side, there is now the possibility that it may receive duplicate notification messages. This issue is discussed in the next section.

There is also a possibility that a notification message will be sent, but it will be received after a client application has already initiated a transaction involving the data for which the notification was sent. In such cases, the current transaction is invalid. More specifically, if the client transaction's ROS or MOS contains any objects for which there exists an unread notification message, the current transaction is not valid by the standards of absolute consistency. Thus, data consistency is preserved only if the application were to abort its current transaction. There must therefore be a mechanism that will check for unread notification messages prior to a commit of any transaction within the entire system.

2.4.2 Duplicate Notifications

Duplicate notification messages do not have the potential to negatively affect the correctness of the system. This is because a client that receives a duplicate notification faces one of two scenarios. If the modified object has not been used in a transaction, the client can simply re−update the value without any invalidation of its current transaction. If the modified object has already been used in a transaction, the application can simply abort its transaction and restart. Clearly, this method is inefficient, as an essentially valid transaction is needlessly aborted, but it does not endanger the correctness of the system.

A simple solution to the aforementioned inefficiency is to tag each notification message with a unique ID. An application that keeps track of ID's can thus know if a received notification is new or is a duplicate of a previously received message. The case where valid transactions are unnecessarily aborted is thus avoided.

The proper response for receiving multiple notifications about the same object must also be defined. It is possible that a client may receive a notification message about an update to an object, and in the process of acting on this notification, receive another notification about the same object. In some applications, the second notification may be ignored if the updated data does not affect the integrity of the current transactions. If the current transaction depends on the updated data being as fresh as possible, then the application must abort its transaction, and possibly restart with the new updated data. In such cases, the action is the same as receiving invalidation messages.

2.4.3 Wanted/Unwanted Notifications

It is very possible that applications will only desire notification messages during a limited period of time. For example, an application may have many transactions involving a certain object, but may only be interested in receiving a notification regarding that object during a subset of the transactions. To ensure that system does not send a notification message at an unwanted time, the application must be aware of when it is interested in notification, and ignore messages when it is not. It is thus necessary to provide a means for the application to not only request notifications, but also to de−request or "turn−off" the functionality.

One method of achieving this is to explicitly have the application request the notification of specific objects. This requires both the application and the central data server to keep additional state of which objects are notified. When the application is no longer interested in receiving notifications, it can make a de−request call for the objects. Thus, the notification messages are not sent, and the application does not expect any notifications once the de−request is made.

Alternatively, the system may allow for a request for notification to expire. Applications may only want to receive notifications regarding a certain object for a given period of absolute time, as opposed to a subset of transactions. The system can allow the applications to determine the lifespan of notifications that are not permanent. Thus, irrelevant notifications will not continuously be sent to applications that no longer are looking for them. In this case, there is no need for a de−request message. It should be noted, however that in such scenarios, there is a chance that an unwanted notification message may still be sent. It is the duty of the application to appropriately ignore these messages.

2.4.4 Invalidation and Notification

Earlier in this section, the concept of invalidation was discussed. While notification increases the flexibility of the system, it does not altogether replace the need for invalidation mechanisms. Allowing invalidations and notifications to exist simultaneously introduces fundamental functionality issues. For example, if the system allows for an invalidation message and a notification message to be sent for the same piece of data to the same client, the correct action is ill−defined. Assuming the case where a notification message is received followed by an invalidation message, it would seem that a valid transaction (as a result of the updated data sent by the notification) would have to be aborted by the application in light of the invalidation message being received. While this behavior does not cause data inconsistency, it does negate the effects of notification. Thus, clients are allowed to request both invalidations and notification within the same application, but not on the same data.

2.5 THOR

The implementation of notification in this paper uses THOR, a distributed object−oriented database system. This section gives an overview of the THOR architecture, and shows how the system interacts with its multiple applications to preserve global data consistency.

2.5.1 THOR Architecture

The THOR environment encapsulates all data in the form of objects, where each object has a unique identifier, as well as its own set of methods that allow access and modifications. The architecture is in the form of a client−server model that keeps persistent state of each object. Any persistent object is stored at the Object Repository (OR), which constitutes the server side of THOR. The OR contains a root object, through which all objects are reachable. The OR is responsible for checking the validity of any potential transaction, and ultimately committing the transaction if valid. The validity of transactions is determined by an optimistic data consistency protocol.

The client side of THOR consists of a Front End (FE), which serves as a local cache for the application that runs on top of it. There is a one−to−one mapping between FE's and applications. An application's FE contains copies of a subset of the objects stored at the OR. The application accesses objects stored at the FE through its own transactions. A commit of a transaction is requested via the FE, which in turn passes the request on to the OR. The FE and OR communicate through a network connection, over which a set of pre−defined messages can be sent.

If an application requires an object that is not stored at its FE, the FE can request the object from the OR, which will send the object back to the FE. The application can then read and/or modify its objects during the course of a transaction. The application invokes the FE interface to commit a transaction, and the FE will send the commit message on to the OR. If the OR determines the transaction to be valid, it will commit the updated values, otherwise it will return an abort message back to the FE. The application then determines the appropriate course of action in the case of an abort, i., abort or retry. Additionally, the OR can send invalidation messages to FE’s that contain stale copies of objects. Upon receipt of the message, the FE updates its invalid data to the new value. Figure 2 displays the overall architecture of the THOR system.

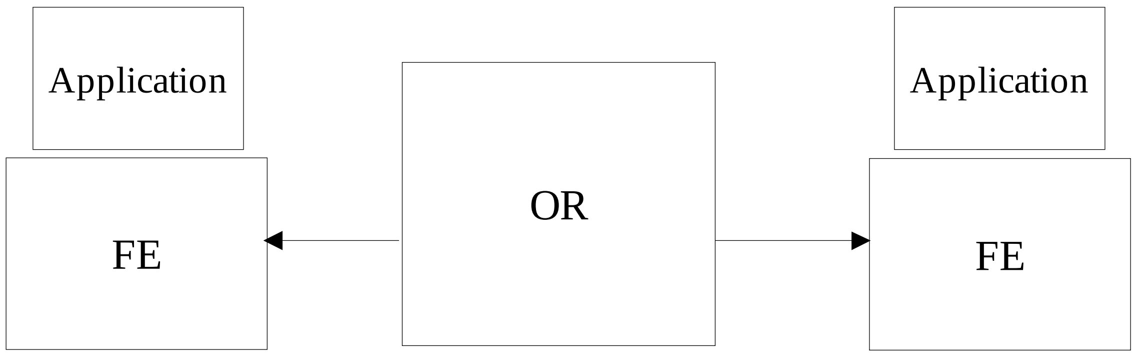


Figure 3 : THOR Architecture

2.5.2 Data Storage in THOR

Every object in the THOR system is assigned an identifier called an *oref* that gives the object a globally unique identity. The oref’s can be used by both the FE and the OR to locate the actual object. The oref is a binary number, which is divided into a page id (pid) and an object location within the page called the Object ID (oid). Part of the pid identifies the OR where the page is stored. This is known as the OR number. The oid’s do not explicitly refer to the location within the page (the page offset), but instead are mapped to page offsets. This allows objects to easily be moved within a page, by simply updating the mapping.

FE’s each have a Resident Object Table (ROT) that converts the oref’s to a location in local memory, through a process called *swizzling*. Thus when an application requests the FE, the ROT maps the oref to the objects handle. The ROT entry, not the oref is used to locate the object. All FE accesses to an object thus go through the ROT. Additionally, the ROT is used by the FE to record object accesses during the various transactions for cache management purposes. This allows the FE to keep track of which objects are read and written during the course of a transaction, which is essential to the FE’s transaction management, which is discussed in section 2.5.3.

As hinted previously, the objects are stored in pages at both the FE and OR. The contents of the pages are the same globally, unless an FE compacts its page to free up space. Thus, when an object is requested from the OR by the FE, the entire page containing the object is sent by the OR.

2.5.3 Transaction Management

Each FE records the transactions of its application in a log that keeps track of the operations performed on the objects. The log records each time an object is accessed. An access consists of either a read from, a write to, or the creation of the object itself. It should be noted that one transaction for an application may actually be several actions at the FE object level, and thus a log is needed to record all the actions associated with a single transaction. Once the application commits its transaction, the FE references the log in requesting the OR to perform the appropriate actions on the objects.

The FE keeps track of an application’s transaction by maintaining a Read Object Set (ROS), Modified Object Set (MOS), and New Object Set (NOS). The NOS is made up of those objects that are created during the course of the transaction. Each set consists of a list of the relevant objects oref’s. In the case of the NOS, the FE has some "free" orefs that it can assign new objects. The members of the NOS are reachable via a reference from some member in the MOS. The application induces the FE to commit, at which point the FE gathers the information from the MOS, NOS, and ROS and sends a commit message to the OR.

The OR will determine whether or not to commit or abort the transaction from the FE. For each FE, the OR keeps track of which objects are invalidated. If the objects from the FE commit message are contained in the set of the FE’s invalid objects, the FE’s transaction is invalidated, and a failed commit is returned to the FE. The application must then decide how to respond to the failed commit.

THOR has an invalidation mechanism in place that invalidates objects of various FE’s. Upon receiving a valid commit from one FE, it sends invalidation messages to all other FE’s that have copies of the committed object cached. Upon receipt of an invalidation message, the FE will check to see if any of the members of its MOS or ROS match the object that was referred to in the invalidation message. If it does, the transaction is invalid, and must be aborted. An aborted transaction is subsequently handled independently by the application.

2.5.4 Messaging between OR and FE

As the FE and OR are connected over a network connection, the communication protocol between the two is based on a set of pre−defined messages. The OR has a message handler that immediately picks up messages from the FE’s. The messages that it can receive from an FE include fetch_object, commit and invalidation_acknowledgment messages. Its action upon receiving the message depends on the message type and the contents of the message itself. The FE, on the other hand, has a message queue that accepts messages from the OR, which must be explicitly checked by the FE. The FE has handlers for messages including the following: send_data, commit_reply, and invalidation.

2.5.5 Why Use THOR?

THOR was selected as the medium to develop the notification mechanism for a few main reasons. First of all, THOR uses an optimistic locking scheme which is critical to the concept of notification. A notification mechanism would not be particularly useful in a pessimistic scheme. Secondly, the object−oriented nature of THOR allows us to control the granularity of the shared data, and thus the size of the data for which we desire notification. Finally, objects in THOR are typed, meaning that we can control what types of changes are made through the object−oriented interface.

3. Design & Implementation

Section 2 introduced the concepts of notification as well as the THOR database system. This section describes the enhancements made to the THOR architecture that allow for applications and users to be notified of changes made to relevant objects.

Implementing notification within THOR consists of modifying and synchronizing the three main parts of the THOR architecture: the OR, FE, and the application running on the FE. The OR and FE use the existing network message infrastructure to accomplish the notification mechanism. Semantically, the FE sends *notification request* messages for certain objects to the OR, while the OR sends the *notification* messages back to the FE upon receiving a commit that changes the object. The application's interface with the FE determines what objects should be watched for notification, and more importantly, when to pick up the notification message from the FE network queue. At this point, the FE must update its value of the notified object.

While the details of the actual notification mechanism described above are central to the theme of notification, the system must also account for the data consistency issues raised by the existence of the notification system. More specifically, as notification messages replace corresponding invalidations, the system must ensure that an application's transaction does not commit if an unread notification message would invalidate the transaction.

3.1 Enhancing OR for Notification

The modifications of the OR to allow for notification can essentially be broken down into four categories. First the OR requires additional state to identify the set of objects for which each FE wishes to be notified. Secondly, the OR provides a mechanism that allows for its various FE's to identify and request notification of desired objects. Next, the notification message is created within the OR that contains information relevant to the notification, including the updated values of the objects. Finally, the OR must implement a method of actually notifying the proper FE's upon receiving a commit that changes the relevant objects.

3.1.1 OR Data Structures

Additional data structures are required to allow the OR to implement notification. The OR keeps track of the FE's that are connected to it through the use of an FE_Manager object. There is a one−to−one mapping between FE's and FE_Managers at the OR. Among the FE_Manager's member variables is a network address for the actual FE, allowing the OR to physically locate the FE. An additional class called Notification_set is created to allow for a collection of orefs to be stored and managed efficiently within an FE_Manager.

Notification_set's are a storage medium of the object orefs for which an FE wishes to be notified. Its methods include add and remove, which handle inserting and deleting orefs from the set. The orefs are hashed to collection numbers that categorize their collection, such that duplicate orefs within the same collection are not permitted. Note that the same oref may exist across different collections. The grouping of orefs into collections serves as a means to allow for groups of objects to be associated with each other at a given point in time. The collection number is monotonically increasing. An FE may want to be notified of a set of objects instead of just one, and the collection number allows for such groupings. A group of objects can thus easily be added and removed. Two sets of Notification_set's are kept at each FE_Manager. The first, called wanted_objs encapsulates those objects for which the FE desires notification. The other, called notify_objs represents a subset of the wanted_objs objects that, through another FE's commit are the objects that have been modified by the current transaction. The notify_objs Notification_set is a temporary storage medium which includes the orefs of objects that are being modified by the current committed transaction at the OR.

3.1.2 Allowing for Registration of Notification

In order to accommodate notification, the OR must provide a means for the FE to first identify the objects for which the application desires notification. An additional message handler for a new class called FE_recv_notify_msg is thus created to handle these requests from the FE. These are *notification request* messages. The FE_recv_notify_msg object contains the FE identifier and an oref that refers to the desired object. The OR uses a listener to immediately pick up any such messages off the network queue. The FE identifier allows the OR to select the proper FE_Manager object to perform the appropriate functions.

When an FE_recv_notify_msg is decoded by the OR, the oref is added to the wanted_objs Notification_set structure of the FE_Manager. This indicates that the corresponding FE should be notified if a commit by another FE changes the objects referenced by the oref. In order to preserve the correctness of the system, the OR checks to make sure that the oref being registered for notification is not also registered to receive invalidation messages. If the oref happens to be registered for invalidation messages, the OR deregisters the oref for invalidation before registering for notification. Note that the notification message is essentially replacing a corresponding invalidation message. Similarly, the OR has a handler for a FE_recv_unnotify_msg message object that will de−register the FE for notification of the object. The interface of the Notification_set structure allows for all the orefs of a certain collection number to be removed, or just a single oref independently. Once an oref is removed from wanted_objs, the OR will no longer send notification messages to the corresponding FE.

3.1.3 Creating the Notification Message

A new message type called OR_Send_Notification_Msg is defined to encapsulate the information required in a notification from the OR to FE. The message consists of two distinguishable parts: a header containing general information for the entire message, and a variable number of per−object segments that have information relevant to each of the modified objects for which the notification message is being sent. The header of the message contains a unique message type identifier, the OR number of the originating message, and a count variable that indicates how many orefs are contained in the notification message. The second part of the notification message contains the newly updated values of the objects to allow the FE to update its object values directly from the message. Sending the new values of the objects corresponding to the orefs in the message saves the time and network capacity needed for the FE to send a request for the updated object and then wait for the object to be sent back from the OR.

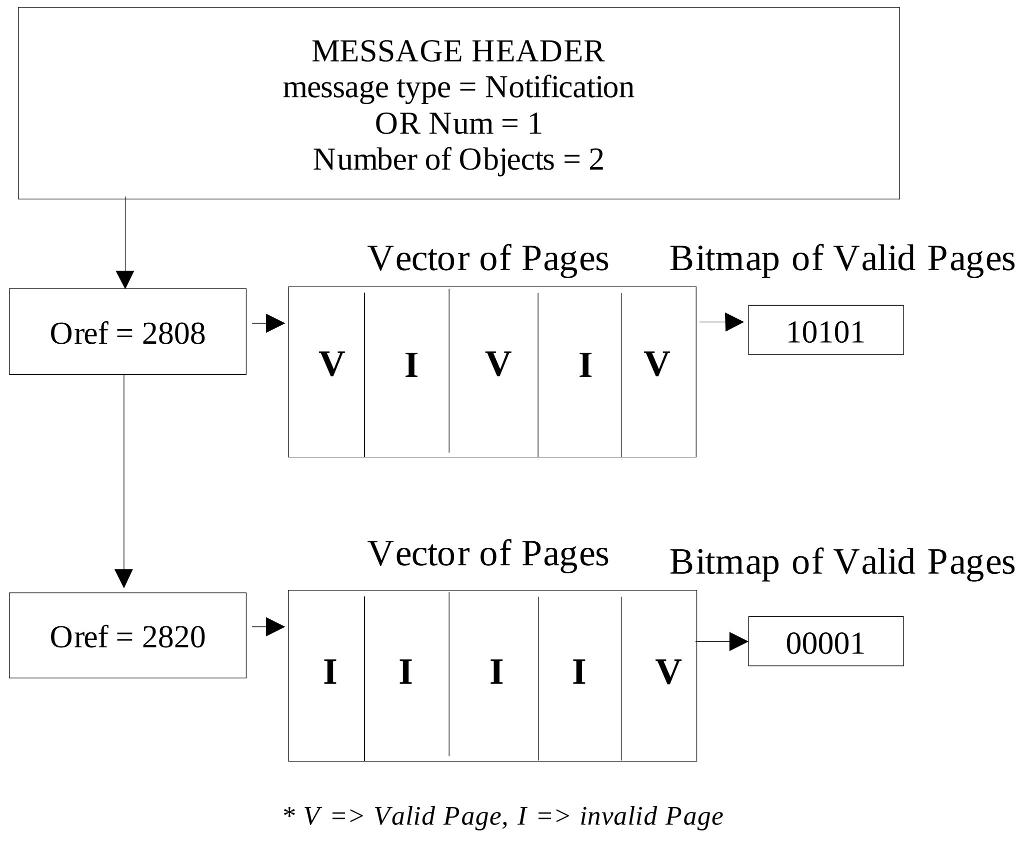


Figure 4: Notification Message for two objects

After the message header, the OR appends per object information for each of the objects that have been modified. This is accomplished by first converting the oref to a segment, which identifies a group of pages. The OR checks which of the segment pages actually contain data relevant to the modified object, and creates a bitmap identifying those relevant pages. The bitmap is later used by the FE to identify these pages upon receipt of the message. A vector containing the aforementioned pages is appended to the message along with the bitmap. A bitmap and page vector pairing are thus appended to the notification message for each object that has been modified. Note that the count in the header of the message equals the number of bitmap/page vector pairs. We will explore how the FE handles the OR_Send_Notification_Msg in a later section.

3.1.4 Notifying the FE upon Commit

Once the OR has registered the FE’s selected objects for notification and is aware of which objects should trigger a notification message, the final step is to send the actual notification message at the appropriate time. An FE_send_notification_msg message type is created to encapsulate the notification information that needs to be sent over the network to the various FE’s. Notification is triggered upon receipt of a commit message from an FE.   The OR’s existing handler for a commit message transfers control to the commit logic within the OR. Here, the validity of the transaction is checked for conflicts that affect data consistency. If the transaction proves to be valid, the modified objects are committed to the OR’s data set, updating the values.

For each oref that is updated by an FE commit, the OR checks to see if any of its other FE’s require notification of that particular oref. This is accomplished by looping through the FE’s other than the one that initiated the commit, and seeing whether the FE is registered for a notification of a change to the oref (as described in 3.1.2). Basically a call is made to its "member" function of the wanted_objs Notification_set to determine this. If the FE is registered for notification of the oref, then the oref is added to the notify_objs Notification_set of the FE_Manager.

Once the orefs have been added to the Notify_objs object set, the FE_managers then proceed to send the notification messages to the appropriate FE’s. Considering that a transaction commit may very likely involve the modification of multiple objects, there is a distinct possibility that notifications for more than one object will need to be sent at one time to the same FE. If the OR were to send individual notifications messages for each object that was modified, the network could possibly be strained by a potentially large number of messages. Instead, the FE_manager combines all the orefs that have been modified into a linked list. This list is inserted into the FE_send_notification_msg message object. Thus, the multiple orefs require only one message to send it to the FE as opposed to multiple messages. The FE_send_notification_msg is sent over the network to the FE, thereby completing the notification process from the OR side. Once the message is sent, the Notify_objs set is cleared.

3.1.5 Notifying the FE: an Alternative Approach

The notification mechanism that was implemented for the purposes of this paper as described above has a potential scalability flaw that may be addressed in future work. Currently, the OR loops through each FE and checks which orefs are registered for notification, before sending out the notification message. The data for notification is essentially stored on a per−FE basis. In a scenario where there are few objects shared across numerous FE's, the performance could potentially suffer. In other words, for each oref that was just committed, the OR must check each of the FE configurations. It seems a more efficient means of notifying FE's would be to design a data structure that stores the notification configuration on a per−oref basis.

An alternative data structure for storing the notification configuration would be to hash each notifiable object oref with a list of FE's that are registered to receive notification on that particular object. Thus, the burden of needlessly checking the configuration of each FE when determining whether to send a notification message is lifted. When an object is committed, the OR need only check the list of FE's already hashed with the oref to determine which FE's need a notification message sent.

3.2 Enhancing the FE for Notifications

The enhancements made to the FE to allow for notification are broken into four segments. First, the FE must have means of creating and sending the *notification request* message to the OR in a manner that is synchronous with and understandable by the OR. Secondly, the FE receives the *notification* message from the OR off the network queue. The notification information for each message should then be locally stored at the FE. Such information is further used to check the validity of transactions that have yet to commit at the the FE. Finally, the FE uses the stored information from the notification message to update the value of its objects.

3.2.1 Sending the Notification Request Message

In order to send a notification request message, the FE creates a message of type FE_Send_Notify, which contains the orefs of the objects the application wishes to be notified of. Once initialized by the application, the FE_Send_Notify message is sent to the OR via the network. The process for deregistering the notification consists of simply sending an FE_Send_Unnotify message to the OR. Once a notification message is received from the OR at the FE queue, the FE must explicitly retrieve the information from the message and act accordingly to update its data values. This functionality is explored in the subsequent sections.

3.2.2 Receiving the Notification Message from the OR

The goal of the FE in retrieving notification messages from the OR is to handle all messages in a way that will preserve the data consistency and correctness of the system. This entails not only updating the values of the modified objects, but also ensuring that transactions involving modified objects are not committed. The latter is necessary to preserve the absolute consistency constraint discussed earlier. There are potentially three situations in which the FE can pick up the notification message: an explicit call by the application to check for notification messages, a waiting mechanism in which the FE waits on the network for any notification messages, and a check for notification in the process of a commit by the application, in which the contents of notification messages are checked to determine whether the transaction being committed is valid.

An explicit check for notification is initiated by the application running on the FE. The application user may initiate the call or the application may automatically make the call without user input. Such a decision is an application dependent issue. A waiting mechanism is also provided by the FE to "wait" for any notification messages to appear on the network queue. Such a mechanism is ideal for applications whose continuity depends on receiving a notification message. Thus repeated explicit calls by the application are not necessary. The application would essentially be idle as the FE waits for any information to appear on the network. As the delivery of any notification message is not ensured, the duration of the wait is some finite value determined via the FE / application interface. The waiting can be interrupted when either a message is received on the network queue, or the application interrupts the wait process. Allowing the wait to be interrupted enables the application to perform transactions and process information.

Notification Message Receipt by FE

Receipt of Notification Message	Transaction Result
During Commit, check queue for notification message	If object for which notification is received is contained in transaction ROS or MOS =>Abort Otherwise => continue
Explicit call to check network queue once for notification messages	Incorporate modified objects into local data set and continue
Wait by listening to network queue. Continue if application interrupts or Notification message appears on network	Incorporate modified objects into local data set and continue

Table 1: Possible Areas of FE Notification Receipt

Absolute data correctness demands that a transaction not be committed by an application if a notification message exists for one of the members of the transactions MOS or ROS. Invalidation messages are used in the absence of notification to enforce correctness. As mentioned earlier, however, notification messages replace all instances of invalidation messages. Thus the notification mechanism must also enforce correctness in the same manner as invalidations. Using explicit application calls to check for notifications and/or waiting on the message queue for notifications does not ensure this constraint. It is entirely possible that a transaction could be created and committed without the queue being checked for notification messages. This problem gives rise to the third and final method of retrieving notification messages. Before a transaction commit is processed by the FE, the network queue is scanned for notification messages. If any of the orefs contained in the notification messages match any of the members of the transaction's MOS or ROS, then the contents of the transaction are invalid. The FE discontinues the commit process before a commit message is sent to the OR. Furthermore, the FE informs the application of the unsuccessful commit attempt and provides orefs that were modified from the notification message.

3.2.3 Storing Notification Information at the FE

Processing the data contained in a *notification message* first requires that the FE provide a storage medium for the information in the message. Furthermore, the FE must read the data off the network in agreement with the manner in which the OR sends the message (Refer back to section 3.1.3 for details). An OR_Recv_Notification_Msg class is thus created. The message class has one header structure which contains a count referring to the number of objects that the notification message modifies, the OR number of the OR that sent the message, and the array of orefs that are modified by the message. Note that this is the exact same information sent in the header of the OR_Send_Notification_Msg.

The remainder of the data is stored in the *body* member variable of OR_Recv_Notification_Msg. The body consists of an array of instances of the *body_part* class. The body_part class stores the information for each modified object contained in the notification message. For each oref that is contained in the notification message, a body_part is created. Its member variables include the segment of the object, the vector of pages mapped to that segment, and a bitmap identifying which members of the segment were actually sent by the OR. The per−object information is read off the network from the OR_Send_Notification_Msg and stuffed into the OR_Recv_Notification_Msg by the FE.

There are situations where a lag period between the reading of the notification message and the actual updating of the object values are necessary. One of the goals of the FE in updating via a notification message is to inform the application when the objects are modified. It is thus necessary for notification updates to take place only when the FE explicitly checks for notification messages. There are other situations in the THOR system that call for the network queue to be checked for messages. In such cases, the FE stores any notification messages in the manner described above, but does not act to update the values until a later convenient time. In order to ensure that data for which notification messages are received are not modified in the lag time between receipt and update, the FE locks all the pages that were sent in the notification message. Any subsequent access attempts to the page result in failure. As will be seen in the next section, the pages are unlocked after the update is completed.

3.2.4 Updating Local Data from Notification Information

After storing the notification message from the OR at the FE, the final step in the notification process is to update the values of the modified objects in the OR_Recv_Notification_Msg object. The header of the message object identifies the number of orefs as well as the orefs themselves. For each body_part instance in the OR_Recv_Notification_Msg object, the FE incorporates the modified pages into its own persistent cache. This is done using the bitmap in the body_part to identify which pages of the objects segment are being incorporated. For each of these pages, the FE checks to see if the page already exists. If it does not, the page number is added as an entry to the page map and then the page is added to the FE's persistent cache. If the page already exists at the FE, the FE merges the old page with the new, keeping the changes within the modified page. Once this has been completed, the lock on the page mentioned in the previous section is released. The update of the locally cached data is now complete

3.3 Application Support for Notifications

The degree of involvement of the application in supporting the notification mechanism varies significantly with the type of application as well as the type of notification the application is interested in.

From a high level perspective, the application identifies those object handles for which it is interested in receiving notification messages. These object handles are converted to the globally recognized oref values. The notification request message takes these orefs and adds them to an outgoing notification request message via the FE interface. Upon receiving a notification message from the OR, the FE updates the object values as described in the previous section. A list of the modified orefs is passed back to the application layer. At this point, the applications use of the modified orefs is dependent on the context in which the application is used. It can inform the user of the changes, or hide the changes altogether. In order to identify which objects were modified from the users' perspective, however, it is necessary to convert the orefs back to the relevant object handles.

Determining which object handles to request notification for is another application dependent issue. If an application is interested in only the value of a single object being modified, the process is straightforward: simply convert the object handle to the corresponding oref and use the FE interface to request notification. There are some situations, however, where the application desires not only a single object, but any new entities related to the object. For example, the calendar application described earlier might request notification for modifications made to a single calendar object. It is reasonable to assume that modifications could include adding a notice or an object. Depending on the THOR object representation of the calendar, an added notice may be assigned a different oref from that of the calendar itself. It is thus necessary for applications to be aware of the object representation of its data set in order to properly request notification messages. This next section explores how the calendar application would be modified to support notification.

3.3.1 Identifying the Calendar Objects to Notify

The Calendar application consists of a collection of calendar objects. Each calendar object is a representation of scheduled events for a given individual. Within the context of the calendar application, we desire the ability to receive notifications of modifications made to individual calendars. The request for notification to a specific calendar is accomplished by deriving the oref of the individual calendar object. This in itself, however, is not sufficient to achieve successful update via notification. An oref is assigned to each calendar object, as well as its member variables. The items in the calendar object that represent each of the scheduled events are stored in an array called "Items", which is one of the calendar object’s member variables. "Items" is itself a pointer to the array. By simply requesting notification of changes made to the calendar object itself, objects added to "Items" will not be included in the notification because although the contents of "Items" have changed, the location of the array has not. Figure 4 displays the object representation of the Calendar object.

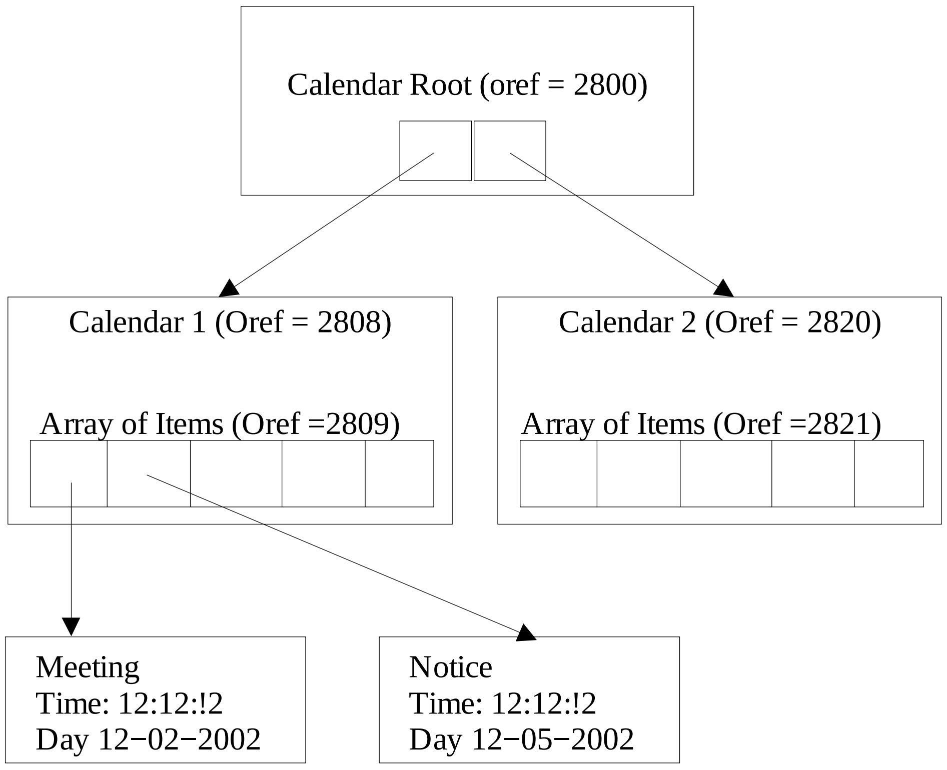


Figure 5 : Object Representation of Calendar Application with two Calendars

It is thus necessary to send the oref of the array "Items" along with the calendar object oref in the notification request message. Figure 4 displays the object representation of a calendar application with two separate calendar objects. Notice that the Items array is assigned its own oref. The next section explains how the application disseminates notification information to the correct calendar object.

3.3.2 Receiving Notification for Calendar Objects

For the sake of the calendar application, it seems reasonable to require notifications of modified objects to be conveyed directly to the user via the user interface. This is accomplished via a notification dissemination mechanism within the application. It is first necessary to identify all the areas within the application that a notification message may b received. As discussed earlier, the application can invoke a wait−for−change as well as a direct request to check for notification messages on the queue. The third area of notification receipt is potentially after a failed commit.

The FE is further modified to keep track of which object orefs were modified by notifications. The interface between application and FE allows for the calendar application to access these modified orefs during the three aforementioned times. Once the list of modified orefs is obtained by the calendar application, the application must distribute the notification to the appropriate calendar objects. This is accomplished by looping through the orefs. For each oref, the application determines which calendar object the oref represents, thus determining which calendar was modified. The calendar object itself is enhanced by adding a boolean field which indicates whether the calendar was modified. If the application determines that list of modified orefs indicates a calendar was modified, the boolean field is set to true.

Whenever a calendar is displayed, the application checks to see if the boolean field is set. If it is, a message is displayed to the user, indicating the calendar was modified via notification. The boolean field is then reset by the calendar. This completes the entire notification process.

4. Related Work

Notification of changes made within a distributed database environment is not a new concept, as several systems have been implemented with notification mechanisms. This section explores some notification mechanisms that are related to the one described in this paper.

4.1 GARDEN Object Server

GARDEN [8] operates on a server that provides persistent and sharable object−oriented data storage and retrieval. Multiple applications can locally cache common data objects. The applications can request "triggers" from the server. An application may have either a read−lock or a trigger on a given object. Semantically, the notification mechanism is similar to the one described in this paper. Once a transaction is committed by an application, the server sends a notification message to all applications holding a trigger on the modified object.

There are some noticeable differences between the THOR and GARDEN systems. First of all, GARDEN runs a pessimistic locking scheme, as locks are used to ensure consistency. This allows for the possibility of deadlock occurring. THOR uses an optimistic scheme in which invalidation messages are used in lieu of locks. Secondly, the notification intricacies differ between the two systems. In GARDEN, one message is sent for each object that is notified within a transaction. The THOR notification system includes all the objects that were modified during the transaction in a single message. Furthermore, GARDEN sends its application only a message indicating the object has been changed. The notification messages discussed in this paper include the updated values of the modified objects.

4.2 ORION

ORION [3] is an object−oriented database system that supports versioning and change notification. ORION supports both message and flag−based notification whereas the implementation discussed in this paper supports only message−based notification. The message−based notification infrastructure in ORION relies on an object representation that differs significantly from that of THOR. Notification messages are sent from object−to−object. When an object is modified, it sends a message to each of the objects that it references. Each instance of an object must therefore have a message handler for each object−type that references. In contrast, notification in THOR uses the existing messaging architecture in which the server (the OR in this case) sends notification messages out to the clients, who in turn modify the objects. ORION was designed with a local environment in mind, as opposed to a globally distributed environment.

4.3 Relational Database Notifications

For the purposes of this paper, we have dealt with notification involving object−oriented databases such as THOR. Notification, however, is also useful in more traditional systems such as relational database management systems (RDBMS). The Buneman and Clemons paper [1] explores the concept of "alerters" in relational database systems. Alerting essentially refers to the notification of users when certain conditions are met. A distinction is drawn between simple alerting and complex alerting. A simple alerter is sensitive to one piece of data being modified, which is similar to the notification trigger described in this paper. A complex alerter may monitor the contents of multiple relations at the same time. The challenges faced in implementing complex alerters deals with placing alerters on relations that may not currently exist, also avoiding costly recomputations of determining whether trigger conditions involving the multiple relations are met.

The increased flexibility of complex alerters allows for a broader functionality in application notification. For example, the following command may be used in a hospitals blood bank database: "Notify if the amount of any blood type falls below a certain threshold." Such functionality is currently not supported by the THOR notification mechanism described in this paper. Allowing for a relational trigger involving multiple objects such as the one in the Buneman paper would require a broader triggering criteria than modifying a single object. Such increased flexibility will be addressed in the next chapter of this paper.

5. Future Work & Conclusions

The notification mechanism described during the course of this paper provide a foundation for future features and applications that make interesting use of the notification architecture. This section explores two such possible works: a criteria−based notification system, and a mobile transaction infrastructure that utilizes the existing notification system.

5.1 Criteria−based Notification

In the existing notification mechanism described in this paper, notification messages for a certain object are triggered upon any modification to that object. Such a triggering mechanism works well if the application awaiting notification is interested in any modifications, such as the calendar application described in Section 3. There are instances, however, when an application or user may only be interested in certain types of changes to an object, or even relationships between different objects. In these cases, receiving a notification message upon any change is not particularly useful, and would seem to burden the network with messages that are not used by the application.

In these situations, having notification messages sent only when the modifications to the object meet certain criteria appears to be a more efficient use of the network resource. This would require that the OR perform the criteria check upon receiving a commit, and then send the notification message if the criteria was met. The object−oriented nature of the data is particularly useful given that objects are type−safe. In other words, by knowing the interface of the object’s methods, the application can reason about exactly what types of changes can be made by a third party. The application can thus explicitly distinguish the types of modifications for which it wishes to receive notification messages.

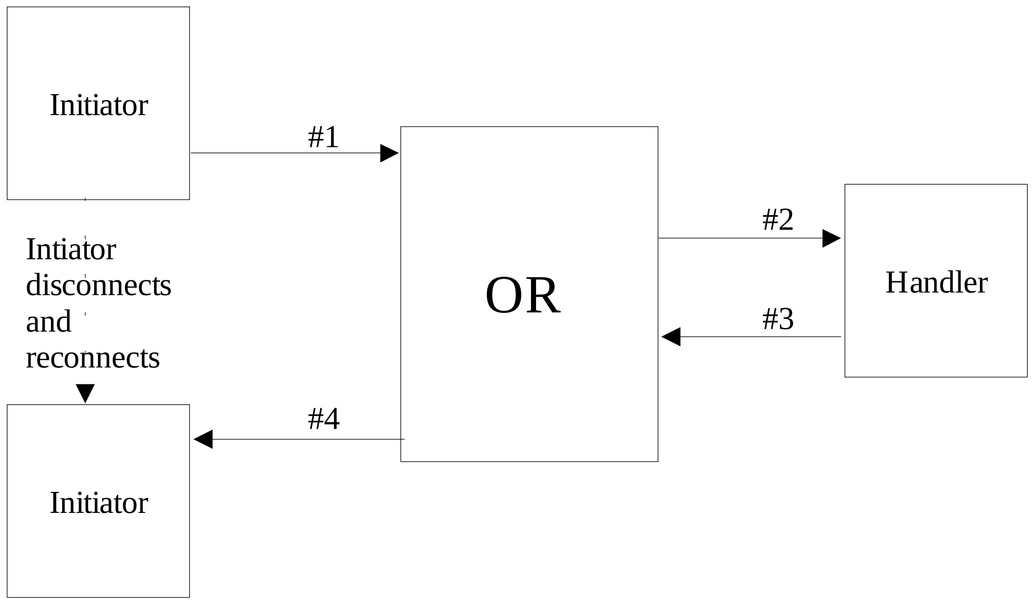
An example of a system that could use such criteria−based notification is a temperature monitoring application that requests to be notified when the temperature value exceeds a certain threshold. The application desiring the notification could send a notification request message to the OR on the temperature object value with the criteria "temperature < 70". For every commit the OR receives from another applications FE that modifies temperature, the value of the new temperature object will be compared with 70. If the new value is less than 70, then a notification message is sent to the OR. Note that in this case, the OR is making use of the temperature object’s method interface to access the value of the temperature.   The OR now sends only messages for which the waiting application is truly interested.

The criteria check could further be extended to send notifications only when certain relations between the objects are met. For example, the previous section, we discussed the following criteria for notification of a hospital blood bank database: "Notify if the stock of any blood type falls below a given threshold." The OR would thus check various objects before determining whether to send a notification message. Allowing for such relational notification triggers would provide a more flexible and useful notification mechanism.

5.2 A Mobile Transaction Architecture Using Notification

This section outlines a potential use of the notification mechanism described in this paper as a means of enabling a mobile transaction architecture. This system assumes that support for disconnected operations exists. The premise of the architecture is that applications, while connected, can "drop off" transactions with the server before disconnecting. The server, in turn would relay the transaction to another application that is capable of completing the transaction. The results of the transaction will then be returned by the server back to the original application upon its reconnection.

A simple example will be used to illustrate the semantics. Suppose that a user, while connected, requests airline reservations from the OR via a mobile device, and then disconnects. This user will be referred to as the "initiator". We will assume that there is a separate FE/application entity (the "handler") that handles such transactions by checking the availability of flights before reserving a seat. The OR, upon receipt of the transaction request from the initiator, would then send a notification message to the handler with the initiator's transaction request to reserve tickets. Once the handler completes the reservation transaction, and commits the results of that transaction to the OR, the OR would notify the initiator with a notification message that included the results of the transaction after the initiator's reconnect.



#1 => Initiator sends notification request message on a transaction object

#2 => OR sends notification message to Handler with transaction object from Initiator

#3 => Handler processes the transaction object, and commits to the OR

#4 => OR waits for Initiator to reconnect and sends notification message to Initiator with results of transaction

Figure 5: Mobile transaction architecture using Notifications

Some modifications would need to be made to the existing notification system in order to accommodate this type of transaction management. First of all, the initiator's initial request message should include the transaction object itself, instead of simply the oref. The transaction object's contents in this case would contain the details of the requested airline reservation. The OR will be triggered to send a notification message to the handler upon receipt of the request, whereas in the original notification system, the OR was triggered to notify upon commits only.

The handler in this case would attempt to perform the requested transaction with another server to see if the transaction is valid. If the transaction is accepted by the handler, its local data would be updated, indicating a reservation had been made. The transaction object would be modified to indicate successful completion, and committed to the OR.

The final step would be for the OR to notify the initiator of the results of the transaction request. Keep in mind, however, that the initiator may or may not be connected. The OR would thus use a listener that waits for the connection of the initiator's FE, and returns the result of the transaction in the form of a notification message. This is a slight modification of the original in that the notification messaging of the OR may be delayed depending on whether the initiator is connected or disconnected.

5.3 Conclusions

We have described a notification mechanism in a distributed database environment that notifies clients of modifications made to shared data. The notification mechanism described in this paper increases the flexibility and efficiency of applications running in a distributed data environment. In a optimistic shared data system such as THOR, it decreases the likelihood of stale data, and consequently reduces the probability that a client's transaction will have to be aborted. We believe the design of the notification messaging, in which the server initiates the notification messages to the client, not only provides the client with fresher data, but also decreases the amount of network bandwidth wasted on polling.

At a higher level of abstraction, the notification capabilities of such a system sets the foundation for a class of applications that can "expect" relevant notifications, such as the mobile transaction architecture described in the previous section. The notification infrastructure can be extended further to allow client applications in a distributed data system to fully realize the benefits of a globally aware environment.

Bibliography

[1] Peter Buneman and Eric K. Clemons. Efficiently Monitoring Relational Databases. In ACM Transactions on Database Systems, pages 368−382, September 1981.

[2] Arvola Chan. Transactional Publish/Subscribe: The Proactive Multicast of Database Changes. TIBO Software Inc, Palo Alto, CA. June, 1998.

[3] Hong−Tai Chou and Won Kim. Versions and Change Notification in an Object Orie−nted Database System. In 25th ACM/IEEE Design Automation Conference, pages 275−281, 1988.

[4] Lori A. Clarke and Peri L. Tarr. A Framework for Event−Based Software Integration. In ACM Transactions on Software Engineering and Methodology, pages 378−421 October, 1996.

[5] Mark Day. What Synchronous Groupware needs: Notification Services. In IEEE, Proceedings of the 6th Workshop on Hot Topics in Operating System, pages 118−122. 1997.

[6] N. H. Gehani, H.V. Jagadish, and O. Shmueli. Event Specification in an Active Object−Oriented Datbase. In ACM, pages 81−90, June, 1992.

[7] Thomas Krsche and Richard Lenz. Funtionality and Architecture of a Cooperative Database System − A Vision −. In ACM Transactions of Office Information Systems, pages 384−390, November, 1994.

[8] Andrea H. Skarra, Stanley B. Zdonik, and Stephen P. Reiss. An Object Server for an Object−Oriented Database System. In ACM Transactions on Database Systems, pages 196−204, January, 1986.