Issues in Object-Based Notification

Paul Kim and Dorothy Curtis
paulhkim@alum.mit.edu, dcurtis@lcs.mit.edu
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA

Abstract

Integrating notification with shared memory applications is an interesting problem. This paper looks at implementing notification in a shared memory, object-oriented, distributed transaction environment.

1 Introduction

As software applications move from stand-alone entities to interdependent components in increasingly distributed systems, software designers must focus more attention on integrating distributed components. A common scenario involves one application waiting for another to update or modify the value of some shared object. An example of this is a flight arrival notification system, in which a user calls in to an application requesting to be notified once the flight arrival time reaches a given threshold. The application, in turn, waits for some third party application to modify the time-to-arrival value before calling the user back. Assuming the application is connected to the rest of the system via a network connection, the question arises as to how the application can be made aware of a change to time-to-arrival.

One possible solution is to have the client poll the system to check whether the value of time-to-arrival has changed. Repeatedly making calls over a network connection to inquire about the value could potentially waste network bandwidth or be expensive. 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. 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. Combining notification with polling improves the latency at which data is refreshed.

We seek to provide the notification mechanism within

The remainder of the paper is as follows: Background on notification and invalidation will be in Section II. Section III will provide an overview of the implementation environment. Section IV will discuss the details and implementation details of the notification mechanism. Section V will explore related work, and Section VI will conclude with possible future projects that may benefit from notification.

2 Background

This section describes the issues and concepts involved in notification. Further, 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.

The discussion of notification for 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 from the database. Notification refers to the update of an application's data with the changes made by another application. Notification is application specific. Each application running on the database may be interested in notification for different subsets of the centralized data objects. In some cases, a multi-cast notification may be appropriate. For the purposes of this paper, however, notifications will use a one-message-per-application approach. The types of applications for which notification is provided for has an impact on the design and implementation of the mechanism. Two distinct types of applications exist for notification: waiting and non-waiting applications.

2.1 Waiting Applications

Waiting applications are those applications that, while waiting to be notified of a change, do not require any other access to or modifications of data. The example of the flight arrival notification system mentioned in the Section I is such an application.

2.2 Non-waiting Applications

Non-waiting applications encompass those applications that actively read and write data, while expecting some local data to be updated. 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. 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 doctor's 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 eliminate the need for unnecessary transaction aborts by providing updated and valid copies of data.

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 frequently that notifications are expected, the less propagation is necessary. A message that propagates to only the application cache results in the cache updating the application's data without the application 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 very 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 conveying an update through its interface to the user. This is needed when changes may impact the user's interaction with his/her application. An application that would need to notify its user is the aforementioned calendar application.

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.

3 THOR: the Implementation Environment

The implementation of notification in this paper uses THOR[3], a distributed object-oriented database system.

The THOR environment provides objects. Each object has a unique identifier and a set of methods for access and modification. The architecture is in the form of a client-server model that maintains the persistent state of each object, see Figure 1. All persistent objects are stored at the Object Repository (OR), which is 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 committing the transaction if valid. The client side of THOR consists of a Front End

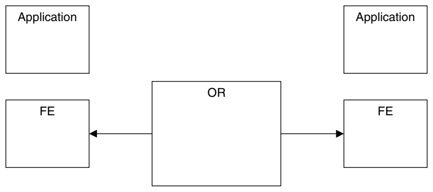

Figure 1. Thor Architecture

(FE), which serves as a local cache for the application that runs on top of it. An application's FE contains copies of a subset of the objects stored at the OR. The application accesses objects as if they were local objects. The FE/OR infrastructure provides, essentially, a shared memory model for accessing the database objects. Commiting changes to the objects is managed through transactions. request on to the OR. The FE and OR communicate through a network connection. 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 the object. 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 is 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.e, abort or retry. Additionally, the OR sends invalidation messages to FE's that contain new values for stale copies of objects.

FE's each have a Resident Object Table (ROT) that maps an object id to a location in local memory, through a process called swizzling.

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. some member in the MOS. When the application decides to commit, 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 stale. If the objects from the FE commit message are contained in the set of the FE's invalid objects, the FE's transaction is aborted, and an abort message is returned to the FE.

The OR uses an invalidation mechanism to inform the FE about stale objects. 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 current transaction is aborted.

4 Design and Implementation

Sections II and III introduced the concept of notification and the THOR database system. This section describes the enhancements made to the THOR architecture that allow applications to be notified of changes made to relevant objects.

4.1 Enhancing the OR for Notification

Additional data structures are required to allow the OR to implement notification. A class called Notification_set is created to allow a collection of object ids to be stored. Notification_set's store the object ids for which an FE wishes to be notified. Two sets of Notification_set's are kept. The first, called wanted_objs contains those objects for which the FE desires notification. The other, called notify_objs represents a subset of the wanted_objs objects that have been modified by the commit of another FE's transaction.

The FE sends an FE_recv_notify_msg to the OR to indicate which objects' modifications are of interest.

Once the OR has registered the FE's requests for notification it is ready to send the actual notification message. At the appropriate time it creates an FE_send_notification_msg message. This message will contain the new value for each object that has been modified and is on the FE's wanted_objs list.

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 object ids 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. 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-object id basis.

4.2 Enhancing the FE for Notifications

In order to request send a notification of changes to a particular object, the FE creates a message of type FE_Send_Notify. This message contains the object ids for which the application wishes notification.

In handling notification messages from the OR, the goal of the FE is preserve 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 aborted. 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. 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. 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. The third place where the notification messages are retrieved is when a transaction commit is processed by the FE. If any of the object ids contained in the notification message match any of the objects in 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 object ids that were modified from the notification message.

The FE stores any notification messages, but does not 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 aborting the transaction. As will be seen in the next section, the pages are unlocked after the update is completed.

4.3 Writing an Application to Use Notification

An application must explicitly request notification from the central data repository for the relevant pieces of data. When designing an application to receive and incorporate notification messages, the responsibilities of the application consistency are twofold. First, the application must regularly check the network queue for the presence of notification messages. Secondly, the application should correctly incorporate the messages. The application must regularly check for notification messages in order to realize benefit from the notification mechanism. If an application were to ignore notification messages until the time of transaction commit, the transaction could potentially be invalidated. For example, an application ”A” may be interested in changes made to a member ”x” of its data set. A notification message informing of a change to ”x” may be sent ”A”, but if the application does not check for the message, then it may continue to perform invalid transactions on ”x”. Upon attempting to commit the transaction, the application will then be informed of the invalidity of its object set. The application must check for notification messages in order to avoid performing invalid transactions, thus realizing a key benefit of the notification mechanism. Upon receiving the notification messages, the application must then correctly integrate the message into its own behavior. If the notification message is for data involved in an uncommitted transaction, then the application must undo the results of the transaction, while also informing the application user of the invalidation if necessary. If the notification message is for data that has yet to be accessed in a transaction by the application, then the application can simply update the relevant data without affecting the validity of its transaction. Again, notifying the end user is an application-dependant issue.

4.4 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 objects for which it is interested in receiving notification messages. 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 objects 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. 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 Notifcation 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 object representation of the calendar, an added notice may be assigned a new oref from that of the calendar itself. The application's calendar class, when requesting notification, must have knowledge of which orefs will change when new objects are added and include those in the notification message. It is thus necessary for applications, or some of their classes, to be aware of the object representations to properly request notification. The next section shows how the calendar application would be modified to support notification.

4.4.1 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 about 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 the array has not.

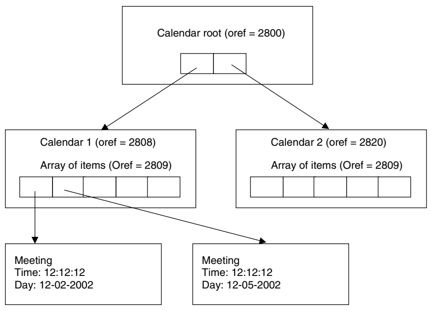

Figure 2. Object Representation of Calendar Application with two Calendars

It is thus necessary to send the oref of the array "Items" along with the calendar object oref in the notification request message. Figure 2 displays the object representation of a calendar application with two separate calendar objects.

4.4.2 Receiving Notification for Calendar Objects

In 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 be 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 objects were modified by notifications. The interface between application and FE allows the calendar application to access these modified. Once the list of modified orefs is obtained by the calendar application, the application must distribute the notification to the appropriate calendar objects.

5 Related Work

Notification is not a new concept. GARDEN [5] provides a server with persistent, shared object-oriented data storage and retrieval. GARDEN uses pessimistic locking while THOR uses optimistic locking. THOR's notification system is more efficient because it includes all objects and their new values while GARDEN's notification indicates that a single object has changed.

ORION [2] is an object-oriented database system that supports versioning and change notification. ORION supports both message and flag-based notification while the THOR implementation 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. 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.

Microsoft SQL Server Notification Services [4] and IBM Everyplace Intelligent Notification Services [1] provide notification mechanisms. The applications that use these systems receive notifications as messages, while, in Thor, the application sees the database objects as local objects, i.e., the Thor applications appear to have shared memory with the OR.

6 Conclusions

We have described a notification mechanism in a distributed database environment that notifies clients of modifications made to shared data. This notification mechanism increases the flexibility and efficiency of applications to run in a distributed data environment.

We are grateful for support from members of the MIT Project Oxygen partnership: Acer, Delta, Hewlett Packard, NTT, Nokia, and Philips. We would like to thank the reviewers for their comments. For readers interested in more information, please see http://nms.lcs.mit.edu/papers/pkim-thesis.pdf.

References

[1] V. Bennett and A. Capella. *Extending Intelligent Notification Services to Monitor New Sources*. `"http://www7b.software.ibm.com/wsdd/library/techarticles/0303_bennett/benne%tt.html"`.

[2] H.-T. Chou and W. Kim. Versions and change notification in an object oriented database system. In *25th ACM/IEEE Design Automation Conference*, pages 275–281, 1988.

[3] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In R. Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628, pages 230–257. Springer-Verlag, New York, NY, 1999.

[4] Notification_Services_Product_Team. *Microsoft SQL Server Notification Services Technical Overview*, 2002. `"http://www.microsoft.com/sql/techinfo/development/2000/sqlnsto.asp"`.

[5] A. H. Skarra, S. B. Zdonik, and S. P. Reiss. An object server for an object-oriented database system. In *ACM Transactions on Database Systems*, pages 196–204, January 1986.