Adapting an Object-Oriented Database for Disconnected Operation

by

Sidney H. Chang

S.B., Computer Science and Engineering
Massachusetts Institute of Technology (2000)

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 2001

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by ....
Dorothy Curtis
Research Staff, MIT Laboratory for Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Adapting an Object-Oriented Database for Disconnected Operation

by

Sidney H. Chang

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Wireless computers are often only intermittently connected to the network. Thus disconnected operation is an important aspect of wireless computing. Disconnected operation focuses on providing useful services to a mobile device even while disconnected from the network. Further, if shared data is in use, disconnected operation must deal with the management of operations on shared data. If the user of a disconnected device operates on shared data, the data must be cached before disconnection. Cached data must be reconciled with the original data once the device is reconnected to the network. Even worse, if two users have cached the same data and changed it while disconnected, this causes a conflict and some form of conflict resolution must be made in order to integrate these changes. In this thesis, the Thor object-oriented database system is extended to support disconnected operation. While many schemes have been devised to avoid conflict resolution or provide mechanisms to automatically resolve conflicts, the approach to conflict resolution taken here is to develop a framework to facilitate conflict resolution by the application. This study provides interesting information about how applications, such as a shared calendar, can be adapted to fit a disconnected transaction model of the system in order to achieve the proper semantics.

Thesis Supervisor: Dorothy Curtis
Title: Research Staff, MIT Laboratory for Computer Science

Acknowledgments

This thesis was made possible by the help of a variety of people right from its very beginnings back when we were reading papers just to find a thesis topic.

The most important person in making this thesis happen was my advisor, Dorothy. Thanks to Dorothy for being an advisor that I could always talk to and for all of her thoughtful guidance throughout the project.

I would also like to thank John, Liuba, and Steve for their help in developing my thesis topic early on. And also thanks to Liuba for helping out with the performance testing in the end.

Thanks to Yan and Chandra for not only helping me understand the intricacies of Thor but also for making it fun. Thanks to my officemates Nick and Peter who made the days in the office all that much shorter. And thanks to the rest of the members of the NMS group especially Bodhi for making lab an interesting place.

Finally, I'd like to thank Andrew for providing the support and motivation necessary for finishing this thesis and also for making the last year at MIT so memorable. And thanks to my parents who have always been there for me throughout my years at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Mobile computing devices such as personal digital assistants, laptops, and cellular phones have become a common part of everyday life. Many of these mobile devices now have wireless capabilities that allow them to communicate with other computers over a wireless network. However, wireless computers will often only be intermittently connected to the network due to expensive wireless networks and concerns over power consumption. As a result, wireless devices must also support *disconnected operation* or the autonomous operation of a mobile device while disconnected from the network [18].

Additional support is required for disconnected operation since even while disconnected, a user may wish to perform operations that would normally require the usage of a network connection. In particular, mobile users may make use of applications that normally access corporate or private databases through a network connection. For example, a travelling salesman may wish to use his laptop to enter a customer's order into a database located in the main office while he is at the customer's site but does not have network access. Another example is a worker commuting to the office who would like to use his hand-held device to enter an appointment into his calendar which is stored in a central database at work.

In both of the sample scenarios described, users of mobile devices are accessing and modifying data that is centrally stored and normally accessed via a network connection. The data in this central store can be accessed and shared by multiple users. A key issue for shared data in collaborative applications is, multiple users may be accessing and modifying the same data *concurrently*. Thus a user may read stale data or one user's updates may conflict with another user's updates to the same data. For example, in the case of the travelling salesman, one user may update a customer's record to reflect a recent sale while another user concurrently updates a customer's record to indicate that that customer no longer has a viable credit standing. In the case of the calendar user, the commuter may enter an appointment for himself while his assistant enters another appointment for him for the same time. From these examples it is clear that conflicts can arise for disconnected users who share data with other users but also that these conflicts are defined by application semantics and vary widely from one application to the next.

Concurrent access to shared data is often solved by employing transactions to achieve serializability of operations and consistency of data across multiple users. However, for disconnected operation, the transaction model is challenged since users of mobile devices will be more likely to read stale data when disconnected and update conflicts will be more frequent since periods of disconnection will allow that device's view of data to diverge from others. From a system's perspective, a solution to these problems means either preventing them from happening or providing support for resolving them if they do happen. To prevent stale reads and update conflicts from happening a system must guarantee that all users always read up-to-date copies of the data. This is often achieved with locking or *pessimistic* schemes where only one user may access a piece of data at a time. But for a mobile setting, this constraint is not acceptable. Requiring that a disconnected device possess a lock on the data in order to update the data either prevents other users from having that data available until the disconnected device reconnects and releases the lock or prevents the user of the disconnected device from performing operations while disconnected. Therefore a system that supports disconnected operation should allow for a non-locking or *optimistic* scheme that allows any data to be accessed. However, the result of an optimistic scheme is that the system must then provide support for disconnected devices to resolve update conflicts since stale data may be read and concurrent updates may occur.

Providing support for resolving update conflicts faces a tradeoff between general support for a large set of applications and providing enough functionality for an application to satisfy its goals. Since as previously stated, conflicts are defined by application semantics, system support for conflict resolution is highly influenced by the application. If a system generalizes support for a large set of applications, it often ends up offloading much of the logic to the application. But if a system provides support specific to a particular application it precludes other applications from using the system since another application may need completely different semantics to resolve its conflicts. The ideal solution is to find a balance in the system so that it provides a sufficient set of tools that are general enough to support a large set of applications but also allows applications to resolve their conflicts easily and without interfering with the consistency of the data in the system.

1.2 Approach to Conflict Resolution in Thor

This thesis describes an investigation of the problem of providing support for resolving update conflicts within the context of disconnected operation on mobile devices. It includes the extension of an existing object-oriented database, Thor, to support disconnected operation and the usage of that system as a platform for examining how applications make use of the conflict resolution mechanism provided by the system. Similar to [9], this investigation attempts to *extract a set of requirements for collaborative applications on mobile devices that use shared data and evaluate the support provided for meeting those requirements.*

There are many possible ways for a system to provide some mechanism for managing conflict resolution ranging from application specified mechanisms to completely system automated resolution (see Section 2.2). We approach conflict resolution in Thor by defining conflicts independently of application semantics and solely as a means of maintaining consistency of objects and leave conflict resolution up to the application. The rationale is that, the system's main job is to maintain the consistency of the data and it will always be the case that the user will be asked to manually repair *some* conflict since the system will not be able to handle all conflicts in all cases for all applications. Total elimination of conflicts is, by definition, an unattainable goal in an environment where concurrent updates to the same data are permitted [14]. In addition, this simple approach to conflict resolution will provide the framework necessary for investigating the needs of different applications in resolving conflicts and for discovering what in addition to the minimum support given, can the system provide to help satisfy these needs.

By implementing disconnected operation in Thor, this thesis aims to provide both consistency of shared data and flexibility in conflict resolution for a wide variety of applications.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Section 2 provides background on issues in disconnected operation (including previous work in conflict resolution mechanisms) and an overview of Thor. Section 3 will discuss the design and implementation of the extension to Thor to support disconnected clients. In Section 4, we discuss an evaluation of the system using a sample application and performance tests with object-oriented database (OODB) benchmarks. Section 5 compares this work with some related works and finally, Section 6 concludes with a summary of the thesis and suggestions for future work.

Chapter 2

Background

In this section we present an overview of the issues in supporting disconnected operation. Then we focus on the issue of synchronizing data upon reconnection and how previous systems have provided support for conflict resolution. This is followed by an overview of the system in which this work is implemented, the Thor object-oriented database system.

2.1 Disconnected Operation

For a system to support disconnected operation there are generally three phases to consider: preparing for disconnection, operating disconnected, and reconnection. Associated with each phase is a set of issues and each issue has a corresponding set of possible approaches. Table 2.1 is derived from Petoura's summary of system-level support for disconnected operation in [18]. It summarizes the phases in disconnected operation and the issues and approaches associated with each.

To prepare for disconnection, the main issue is the prefetching of data into the mobile device since the repository from which the data is normally retrieved will no longer be accessible. Depending on the system, this repository could be a database server, file system server, another mobile device, etc. And depending on the source from which data is fetched and the rest of the system's architecture, the unit of prefetch will change - it could be objects, files, pages of data, operations, etc. What

	Issue	Approach
Prepare for Disconnection	Prefetch unit	Depends on the system
	What to prefetch	User specified Based on history of operations Application specified
	When to prefetch	Prior to disconnection On a periodic basis
Operating Disconnected	Request for data not prefetched	Raise an exception Queue for future service
	What to log	Data values Operations Timestamps
	When to optimize the log	Incrementally Prior to reconnection
	How to optimize the log	Depends on the system
Reconnection	How to integrate log	Replay log
	How to resolve conflicts	Use application semantics System automatically resolves User specified

Table 2.1: Phases associated with disconnected operation and their corresponding issues and possible approaches.

and when to prefetch is a design decision which impacts the control of the user over prefetching.

What to prefetch can be based solely on a user's history of usage, application specifications, direct user input, or a combination of some or all of these. A user's history of usage can be used in a similar fashion to standard caching mechanisms for data with temporal locality. Usage of history information can be organized on a per application basis or on a total set of operations. For example, in a database system, an application may have a history of accesses to a particular set of data associated with that application and data associated with that application can be prefetched. On the other hand in a file system, the system can track overall usage of files regardless of which application actually accessed the files and prefetch frequently accessed files over all applications. To achieve spatial locality, application or user input may be necessary in order to prefetch objects that are likely to be accessed together. For example, in a calendar application, the system may be able to tell that a certain set of events are of interest to a user by a history of accesses but only the application would be able to tell that since events from one user's calendar are being accessed, all of the data associated with that user's calendar should be prefetched (not just the recently accessed events). Other considerations in prefetching data include information that can not be collected without direct user input such as what applications the user intends to use while disconnected. Yet another consideration is whether or not certain data will be used concurrently by other users. In certain cases it may be better for a mobile device to not prefetch data that may be used concurrently in order to avoid having to resolve conflicts upon reconnection.

When to prefetch data should complement the strategy for deciding what to prefetch. If data is prefetched in the background on a periodic basis it would not make sense to query the user for what to prefetch each time. It would be more suitable to have an application specify what to prefetch in a configuration file or base prefetching on a user's history of accesses. If data is prefetched just prior to disconnection, then it would be suitable to query the user at that point. Although configuration files and history of access information can also be used at that point, history information would likely fetch data that had been useful to the user only directly prior to disconnection and may not have any relevance to what the user would like to use while disconnected.

While disconnected the main issues are what the user is permitted to do and logging. Since the user contains a local copy of the data while disconnected but the repository of data from which it retrieved its copy can still modify that data while the user is disconnected, the first question to deal with is whether or not the user can modify data that may potentially be modified by someone else. In an *optimistic* scheme, where users are allowed to modify shared data that may be concurrently accessed by other devices during the device's period of disconnection, some form of conflict resolution is required - this means that upon reconnection, certain operations made while disconnected may fail and these failures have to be handled. Since this project deals specifically with resolving update conflicts upon reconnection, the user is permitted to modify prefetched copies of data while disconnected. Other *pessimistic* schemes may be devised where a disconnected device will allow only reads of copies of shared data and no writes or a disconnected device can be granted a lock on a piece of data so that only it will have the right to modify a piece of shared data (in which case the user can modify only shared data for which it has a lock while disconnected). The benefit of these last two schemes is that there will be no need to deal with conflicts upon reconnection. However, for the mobile computing setting, mobile users will want to be able to access data even while disconnected so availability through an *optimistic* scheme is granted at the price of having to handle conflicts on reconnection.

Another issue in operating disconnected is if a user attempts to access data that has not been cached. The system may either inform the user that the operation can not be processed at the time and abort that operation or the system can queue the operation to be processed upon reconnection. The user or application may also wish to have input into this process because it may be acceptable to abort an operation and continue with other operations in certain cases but may not be acceptable in other cases. For example, the travelling salesman can not proceed with a customer's order without verifying his credit record but a calendar user may be able to continue adding events to his calendar even if one event was not accessible.

To keep track of the operations that a user makes while disconnected, it is necessary to maintain a log. The log must keep track of the updates that users make while disconnected. The form in which these updates are recorded can be actual data values, operations, or timestamps. Storing operations gives greater context for the modifications that occur while disconnected. This information can be very useful in resolving update conflicts that occur upon reconnection. Timestamps consume less space at the disconnected device but may also the cause reconnection process to take a longer amount of time. Storing data values will make restoring states easier if an abort should occur, but it will also consume more space on the disconnected device. The format in which data is stored in the log will also be highly dependent on the rest of the system.

Since the mobile device will have limited resources and quick reintegration of the log upon reconnection is desirable, optimizing the log in terms of when and how is important. Optimization of the log will depend on the implementation of the log. The log can be optimized incrementally during the disconnect or can be done upon reconnection.

Upon reconnection, the main issues are processing the log and dealing with failures. Processing the log consists of replaying the contents of the log. This process can have different semantics in different systems. The log can be guaranteed to replay in the same order in which it was filled or it can have looser semantics where dependent operations will be in order and others can can be out of order.

The process of replaying the log is often called reintegration or synchronization of the disconnected device's local data with another repository of data. The difficulty that arises is, if in the process of synchronizing the data, a concurrent update to some data in the log has occurred. The problem of synchronizing data between a mobile device and a stationary host is not new. Palm pilot devices are well-known examples of mobile devices that synchronize their data with a desktop computer. However, this work differs from the Palm pilot model because with the Palm pilot the user is synchronizing data only between his own copies of the data meaning that he is the only one really modifying the data. This work investigates support for collaborative applications where multiple users will often access the same data concurrently.

As stated in section 1.1, a common way to handle concurrent accesses to shared data is to use transactions to achieve serializability and consistency of operations on data. Traditional properties of transactions are: atomicity, consistency, isolation, and durability (ACID). These properties need to be redefined, i.e. relaxed, in order to provide transactions in mobile computing. A new transaction model is needed since transactions can no longer commit or abort at the ending boundary without connectivity. There can be scenarios in which a mobile device is granted "ownership" of a piece of data and so it can use normal transactions on that data, however this would prevent other users from being able to access that data. Various transaction models for mobile computing have been studied in [17, 8, 16]. Each is similar in that *weak*, *tentative*, or *second-class* transactions are defined for transactions made on data local to a mobile device while disconnected. They are considered as such until the device reconnects with the repository of data and synchronizes its tentative transactions to become persistent or permanent transactions.

To deal with conflicts or aborts that occur upon reconnection, there can be several approaches. The systems can automatically try to resolve conflicts. This may be possible for systems where the semantics are simple but for systems where a variety of applications with different semantics can be making use of the system, it may be too difficult for the system to automatically resolve the conflicts in some reasonable manner for every application. Another way to resolve the conflicts is to consult the user upon a failure. However, this may be cumbersome for the user and would require giving the user enough context in order to understand why a failure occurred in order to have an idea of how to resolve it. Conflict resolution can also be left up to the application since it is best aware of its own semantics. However, this can also become difficult for the application programmer. A combination of system, application, and user support can be used to resolve conflicts. The main question is then what is the basic support that a system should provide to the application in resolving conflicts? The next section describes the approach to conflict resolution in several different projects.

2.2 Approaches to Conflict Resolution

Conflict resolution can be approached in different ways. One is to simply avoid it by providing mechanisms for an application to give "hints" to the system for allowing certain concurrent actions or relaxing the ACID semantics of a transaction. Another way is to provide a mechanism for an application to write type-specific procedures for resolving conflicts with a last resort of consulting the user when the resolution procedure fails.

2.2.1 Conflict Avoidance

Avoiding conflicts means allowing concurrent accesses to data to occur in a manner that still results in serializable and consistent results. One common way to allow for concurrency is to make use of commutativity properties of data resulting in a more dynamic atomicity [22]. Commutativity is a property of operations on specific types of objects that allows operations to be reordered and arbitrarily interleaved. Consider, for example, an object that is an integer and it has an increment operation. If two concurrent increments occur their ordering does not matter since either ordering of the increments will still result in the same final integer value. Other more advanced means of allowing concurrency include escrow based techniques, recoverability, and fragmenting objects.

Transaction escrow methods [11] are based on a quantity such as the the total number of items in stock, which is normally locked for the duration of a transaction that reads and modifies the quantity. For example, the transaction could include placing an order and decrementing the number of items in stock. Instead of placing the lock on the quantity for the duration of a long-term transaction, the quantity is locked for only a short time to update an escrow log which has the quantity's state after a number of uncommitted transactions. So, in the course of deciding whether or not to complete an operation, the object's current state and also the state in the escrow log is used to make a worst-case decision (worst-case is if all of the uncommitted operations commit). This has the benefit that locks are much shorter and concurrent accesses will not have to wait long for a lock.

Recoverability [2] provides more concurrency than commutativity. An operation b is recoverable with respect to operation a if the results are the same regardless of whether or not operation a executes before b. For example, a stack push is not commutable yet it is recoverable. So two noncommutative but recoverable pushes could be allowed to execute concurrently, however if both should commit, they would still need to be in a serializable order but if one should fail, the other can still commit since they are recoverable.

Fragmenting of objects [21] is where objects are broken up among disconnected devices so that a disconnected device can operate autonomously on a fragment of an object. Fragmentable objects require that application semantics can be exploited in order to define *split* and *merge* operations that describe how an object can be split into fragments and then how fragments can be merged back into a single consistent copy. Objects that are fragmentable are often aggregate objects such as stacks, sets, or queues.

2.2.2 Conflict Resolution

Rather than try to accommodate concurrency, other systems have tried to accommodate the resolution of conflicts *after* they have occurred. These mechanisms are built into the system and allow for an application to define their resolution procedures based on application semantics. Examples of such systems are Coda, Bayou, and Rover [13, 6, 12].

Coda is specifically designed for file systems. It uses an optimistic concurrency control scheme meaning that any client can update its copy of a piece of data. Coda automatically resolves concurrent updates to directories and also has a mechanism for transparent resolution of file conflicts called *application-specific resolvers* [14]. While the application-specific resolver works in some cases, in cases where it fails, the user will have to resort to interactive repair.

Bayou also uses an optimistic concurrency control scheme and, rather than supporting transactions, propagates updates by pairwise anti-entropy sessions [7]. Bayou is different from other shared data management systems that support disconnected operation because it caters to the application and has many provisions for letting the application specify as much information as possible to help the system determine how to deal with update conflicts automatically [20]. It has provisions such as *dependency checks* and *merge procs*. These can be assigned per object type and are invoked on each update. The dependency check allows the application to specify what integrity constraints must be maintained in the database by an update while the merge proc attempts to handle a failure of the dependency check. However, like the application-specific resolvers in Coda, Bayou's merge procs can also fail in which case the application or user will be consulted for manual repair.

Rover is very different from Coda or Bayou. It essentially provides a framework for queued remote procedure calls between a disconnected client and a central server.

Client applications cache objects from the servers and queue operations made on those objects while disconnected [12]. Rover provides clients with facilities for conflict detection and resolution, however unlike Coda or Bayou, the system itself has no notion of problems arising from concurrent updates to the same object. The application must specify what a conflict is, even in the case of two clients updating the same object. However, Rover's model of passing operations rather than only new data values, allows an application to have the context of a failure and also allows an application to make use of commutativity of objects.

2.2.3 Summary of Approaches

Clearly there are many different mechanisms for conflict prevention and resolution. However, prevention mechanisms work only for specific cases and making use of such mechanisms can be hard. For example, escrow methods are appropriate only for numeric values and defining split and merge operations for fragmentable objects may not be immediately obvious for more complex data types. In addition, resolution mechanisms in systems such as Coda, Bayou, and Rover will not be able to handle all conflicts and the user will be asked to manually repair some conflict for some application. Therefore the rest of this thesis will describe the design and implementation of a system that leaves conflict resolution completely up to the application. By investigating applications using this system we attempt to find basic requirements for system support for conflict resolution for disconnected devices.

2.3 Thor Overview

This study of conflict resolution for disconnected devices makes use of the Thor distributed object-oriented database system. This section gives an overview of the Thor architecture and how applications interact with the system to achieve transactional semantics.

2.3.1 Thor Architecture

Thor provides a persistent store of objects where each object has a unique identity, set of methods, and state. The system has a client/server architecture servers are repositories of persistent objects. The OR consists of a root object where all persistent objects are reachable by the root object. The OR handles validation of transactions across multiple clients by using an optimistic concurrency control scheme described in [1]. Clients in Thor are made up of a front end (FE), application schema, and user interface. The FE handles caching of objects from the OR and runtime transaction processing. Applications operate on cached objects at the FE inside transactions boundaries and request to commit transactions through the FE at the OR. Figure 2-1 shows the organization of the FE, application schema, user interface, and the OR. All communication between the client and server is between the FE and OR. All

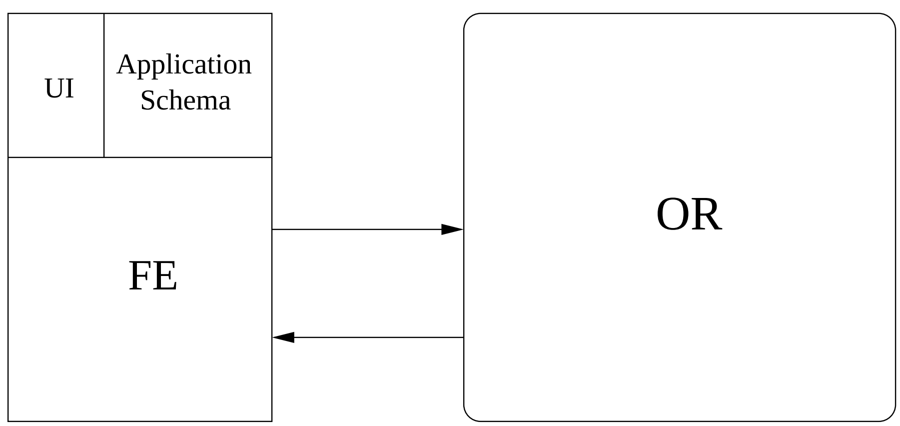

Figure 2-1: Thor has a client/server architecture. The client is made up of the front end (FE), application schema or object definitions, and a user interface. The server is the object repository (OR).

communication between the clients and servers in Thor is between FE's and OR's. The communication is limited to a set of specific messages. Table 2.2 lists the types of messages passed between the FE and OR. The FE will send the OR a root fetch message on startup of the client in order to get a handle on the root of the repository of objects. All other objects in the repository are reachable via this root. During the course of a transaction the FE may request objects by making fetch requests to the OR. At the end of a transaction the FE can send a commit request to the OR. In addition the FE can also request the OR for the allocation of space for newly created objects. The OR can send to the FE invalidation messages when other FE's have committed changes to objects in the repository. The FE must respond to these messages by invalidating stale objects indicated in the invalidation message.

Messages Sent	
FE $\longrightarrow$ OR	OR $\longrightarrow$ FE
− get root	− send root
− fetch object	− send data
− commit request	− commit reply
− invalidation ack	− invalidation
− allocate request	− allocate reply

Table 2.2: Messages sent between the FE and OR are limited to the above. The FE initiates root object requests, object fetches, transaction commits, and allocation requests for new objects. The OR initiates invalidation messages.

2.3.2 Objects in Thor

Thor OR's store objects on disk in pages. FE's cache entire pages of objects and pages will remain identical to those at the OR unless a page is compacted at the FE to free up cache space. A method of hybrid adaptive caching or HAC is used to compact *hot* or frequently accessed objects into a page while discarding *cold* or less frequently accessed objects [5].

Each object is uniquely identified by an identifier known as an *oref*. An orefs is a 32-bit identifier that uniquely identifies an object. It is also used to locate an object within pages at the OR and FE. 22 bits of the oref are a *pid* or page id which identifies the object's page and 9 bits of the oref are an *oid* or object id which identifies an object within a page but does not encode its location. Instead oid's are mapped to offsets within the page in an offset table stored with each page. This allows for easy movement of objects within a page.

At the OR's objects are known only by their orefs. However, not all objects at the FE are persistent. Objects at the FE are categorized as either persistent or non-persistent. Persistent objects are objects that the OR's are aware of and that are reachable from the persistent root object. Non-persistent objects are objects that are newly created by an application that have not yet been committed at the OR or objects that are temporarily used by the application but do not need to persist across different runs of an application. Persistent objects at the FE are stored in the persistent cache which caches whole pages from the OR. An object in the persistent cache can be reached at the FE via its oref. Non-persistent objects, by contrast, are stored in the volatile heap and do not have orefs until they are committed and assigned oref's by the OR.

For a program to access an object at the FE in the persistent cache, orefs must be mapped to local memory. To speed up this lookup process, objects at the FE are *swizzled* so that oref's become pointers into the ROT or Resident Object Table. The ROT has handles to persistent objects so that when a reference is made to an object, rather than use its oref, a ROT entry is used to locate an object. The HAC scheme mentioned above uses the ROT to keep track of accesses to objects and also handles cleaning up unused references in the ROT [5]. In addition the ROT is used to maintain handles on non-persistent objects since they do not have orefs. Note that this means that non-persistent objects can not be removed from the ROT until they have been made persistent or there are no references to them. However, persistent objects may be removed from the ROT if their pages are being removed from the cache as long as there are no references to them.

2.3.3 FE Transaction Logging

Transactions are logged as operations are made by an application within transaction boundaries. This logging is used to determine which objects were read, written, and created within a transaction. All transaction logging is done at the FE as applications operate on objects. Every time an object is read, written, or created it is logged so that the appropriate information is available for when the application requests a commit.

Applications make high level operations on objects as defined in the schema for the application. These high-level operations on objects are reduced to reads, writes, and creations of objects. Each of these is recorded by the FE in order to create the correct information to be sent to the OR in a commit request. While application operations are defined in the schema in Java, the schema are converted to C code which then includes extra code for each objects access that logs the access with the FE.

2.3.4 Committing Transactions

An application can complete a transaction by making a request to the FE to commit the transaction. The FE processes this request by taking all of the logged information on what objects have been read, written, and modified and creating commit sets - the read object set (ROS), modified object set (MOS), and new object set (NOS). These sets are sent to the OR in the form of a commit request. The ROS and MOS will contain only persistent objects and the NOS will contain only those non-persistent objects created inside the transaction that are reachable from some persistent object. Therefore any object in the NOS must be also be referenced in the MOS of the same transaction. The ROS and MOS must have objects in their unswizzled form, i.e. they should only be identified by their orefs and not ROT entries since the OR understands only orefs. Before a NOS is sent to the OR it must contain orefs. The FE maintains some free pages for new orefs but in the event that there are no free orefs available, the OR is contacted to obtain new orefs.

To handle concurrency, an OR will validate a transaction based on whether or not that transaction contains invalidated objects for the FE from which the request came. The OR maintains per-FE a set of invalidated objects which are objects whose state is no longer valid since the time the FE cached them. Objects become invalidated when one FE successfully commits a transaction. Any objects modified by that transaction will be invalidated for other FE's that have cached that object since they now have a stale copy of the object. FE's are notified of invalidations and must acknowledge them by invalidating the objects in the persistent cache so that if those objects are ever accessed by the application, their new state will be fetched from the OR.

The OR can either commit or abort the FE's commit request. If the transaction is aborted by the OR, the FE must then roll back any of the changes made by the application. This includes reverting the state of modified objects back to the original state prior to the transaction and removing any newly created objects from the volatile heap. If the transaction is committed by the OR, the FE will move any newly created objects from the volatile heap to the persistent cache.

2.3.5 Why Use Thor?

After having described an overview of the Thor system, the question of why Thor was chosen for this work may arise. There are several reasons for using Thor:

- To achieve consistency in the presence of disconnected operation, it has been stated that optimistic schemes are more desirable. Thor already provides an optimistic concurrency control scheme.

- To achieve flexibility in conflict resolution, Thor's object-oriented design may provide the flexibility necessary to support a variety of applications.

- A preliminary study of disconnected operation in Thor [10] suggested that disconnected operation would fall naturally into the design of Thor. However, this has not yet been substantiated by any implementation.

Chapter 3

Design and Implementation

In the previous chapter, Thor was introduced. However, it did not support disconnected operation. In this section, the design and implementation of disconnected operation in Thor is described in detail.

A disconnected client in a database system is unable to communicate with a server. Hence, the client can not fetch data, make requests to commit transactions, or receive updates (i.e. invalidations in Thor). To prepare for disconnection, the client can retrieve data that might be useful to a user while disconnected from the server. During the period of disconnection from the server, the client saves enough information per transaction in a log so that upon reconnection with the server, the server is able to commit or abort each saved transaction and the client is able to resolve aborts or conflicts that may arise. When the client reconnects with the server, it synchronizes its log of saved transactions with the server and updates the state of the cache to reflect the commit or abort of each logged transaction and any updates received from the server.

The extension of Thor to support disconnected clients has two main aspects: extensions to the application (user interface and the application schema) and extensions to the FE (caching and per transaction processing). The reason for the division is that certain requirements for disconnected operation are heavily based on application semantics. For example, the objects that should be prefetched into a cache before disconnection depend on what operations a user will be likely to invoke while disconnected. Another example is the resolution of aborts upon reconnection with the OR. If a saved transaction is aborted after reconnection, the application semantics will dictate how the abort should be dealt with.

The following sections will first describe the design of FE support for disconnected clients that is independent from application semantics and then will describe the design of how application based support for disconnected clients should be organized in applications built on top of Thor (a specific example of an application will be discussed in 4.1).

3.1 FE Support for Disconnected Operation

FE support for disconnected operation can be divided into three phases. The first is preparation for disconnection which deals mainly with prefetching of objects. The second is operating disconnected which handles a different transaction semantics for disconnected operation (as opposed to when the FE is connected). The third is reconnecting with the OR which includes the handling of pending transactions and the commits and aborts resulting from them.

3.1.1 Preparing for Disconnect

To prepare for disconnection from the OR, the FE will need to prefetch objects into the cache by processing queries specified by the application. No additional FE support is necessary for this since a prefetch query will look to the FE like any other fetch query. However, to the application the prefetch query will be very different than a fetch query since it may require user input or special purpose queries in the application schema. This will be discussed in section 3.2.1.

A desirable trait of a prefetch query is to not only have the requested object fetched from the OR but also related objects. This is a feature that the FE could implement since the application specifies only a single object that it would like fetched and it is up to the front end to request any additional objects based on locality of objects in the cache. However, this is also already a feature of every normal fetch since an application requests only a single specific object, but the fetch request to the OR will actually return an entire page of objects. This page will likely "have some locality within it and therefore other objects on the page are likely to be useful to the client" [15]. For the FE to provide a greater control over the locality of objects, it is possible for the FE to infer the locality of references made by the application prior to disconnect from usage information stored in the cache. Previous work has also been done on this in Thor with hybrid adaptive caching (HAC) techniques that store hot objects and discard cold ones based on frequency of access [5]. However, this only gives temporal locality information and requires that the FE has already collected such information based on the operation of the application. For spatial locality information, only the application can specify this information since it is aware of application schema and how objects reference one another in different operations. The usage of this information in prefetch queries will be discussed in section 3.2.1.

In addition to preparing the client for disconnection, the OR must also prepare for a client's disconnection. The OR must keep per-FE information in order to know which object invalidations a client has seen or not seen. Invalidations are used in order to ensure that the client has the correct versions of objects. It is important for the OR to have the correct set of invalidated objects per FE. For example, if the OR incorrectly thinks that an FE has already seen an invalidation message for a particular set of objects when in fact the FE has not, it may allow a transaction based on stale data to succeed. While this was not implemented for the system since the focus was on support for disconnected clients at the FE and application levels, possibilities for its design are discussed here.

There are two options for the OR to save state for an FE once it has disconnected. The FE can maintain a timestamp associated with the last information received from the OR. This gives the OR enough information to know the state of the objects the FE last saw before disconnecting. The other option is to have the OR assign an id to the FE before it disconnects and then maintain FE state while it is disconnected including a set of invalidated objects. But the FE may never reconnect so the OR will also have to time out after some finite amount of time and assume that the FE will never reconnect after that point.

The first option is appealing since the OR does not need to maintain any state and does not have to impose a time limit in which the FE may reconnect. However reconnecting with the FE will take much longer since without an invalidated object set and only a timestamp, each object in the write set of a commit will have to be checked against the timestamp of the version of the object at the OR and the timestamp that the FE disconnected with. And then to prevent having to make this check for every transaction thereafter or until the FE has accessed each object at least once again, the entire FE cache must be flushed after emptying the queue of pending transactions. Alternatively, upon reconnection, the OR could recreate an invalidated object set for the FE based on the timestamp but this would require iterating through the entire object repository at every reconnect.

If we expect that the FE will reconnect often, then the second option is a more viable one since the time limit on reconnect can be set with the expectation that the FE will reconnect within that time. Even though the OR will use memory to maintain state for the FE while it is disconnected, it is likely FE's will be disconnecting and reconnecting often enough that it will not run out of memory.

A combination of the two options can be used to allow clients to specify whether or not they will be reconnecting again soon in the future (where the definition of "soon" is defined by the time limit imposed at the OR) or not. Then the appropriate method can be picked by the OR for saving the FE's connection state.

3.1.2 Operating Disconnected

Once the client has disconnected from the OR, an application will still attempt to commit transactions as it normally would while connected except the semantics of the transaction change. While disconnected, a commit becomes a *tentative commit* meaning that the commit could potentially be aborted by the OR upon reconnection.

The first question is then, what do the changed semantics of a commit mean for the state of objects at the FE cache? One option is to change the state of objects in the FE cache to reflect the modifications of a transaction as it is tentatively committed. Another option is to not change the state of the objects in the FE cache to reflect a tentative commit but rather save only the modified state in a log of modifications. The first option follows naturally from the model taken by connected Thor since transactions in connected Thor will modify the cached objects and then revert the changes if the transaction should abort. It makes accessing objects during a transaction easier but makes handling aborts more difficult. The second option is on the other hand the opposite - it makes looking up objects during a transaction harder but makes handling aborts simpler. In the second option, each modified state of an object is saved in a log rather than directly modifying the version of the object in the cache. This makes aborts easy since nothing in the cache is modified. But during disconnection, the application will likely have more than one transaction. So in order to lookup the current state of an object, the FE can not simply look up the state of the object in the cache but must query the entire log of modifications made while disconnected to see if there is a more current version of the object. Therefore the first option is more viable since aborts are expected to be infrequent and since performance at the FE while disconnected should be comparable to when it is connected.

When an application commits during disconnection, a *tentative commit* is interpreted by the FE as logging a tentative transaction in the tentative transaction log. This log saves enough state per transaction made during disconnection in order to replay each transaction once the FE is reconnected with the OR. The application will be given an id upon tentatively committing to identify an operation with a tentative commit which can later be used to aid in resolving an abort. The following sections will describe the details of the log and the state maintained for each tentative transaction.

Tentative Transaction Log

The log is a queue of tentative transactions. When an application makes a commit request, the FE will log a tentative transaction in the tentative transaction log that encapsulates all of the information necessary to process the commit at a later time. Each tentative transaction is numbered from 1 to n where n is the number of tentative

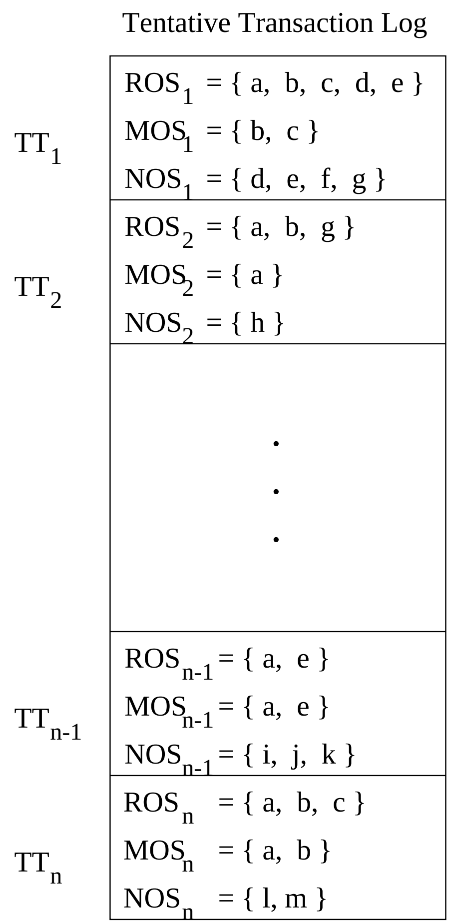

Figure 3-1: Transactions committed while the client is disconnected are saved in a tentative transaction log. Each tentative transaction in the log consists of a read object set (ROS), modified object set (MOS), and new object set (NOS).

transactions in the log. Figure 3-1 depicts an example tentative transaction log with tentative transactions 1 through n labeled as TT_1 to TT_n. Tentative transactions are logged in the order in which they are committed by the application so TT_1 was the first transaction to be committed while disconnected and TT_n was the last.

Tentative Transaction

The model used in connected Thor has the FE maintain a transaction log per transaction. This keeps track of objects read, modified, and created during the transaction. When the application commits the transaction, during connected operation this transaction log is directly translated to a commit request to the OR. But while disconnected, the transaction log is converted to a tentative transaction which is then saved in the tentative transaction log. A tentative transaction consists of commit sets and the saved states of modified objects.

Commit sets are sets of object references and data to be sent to the OR in a commit request. There are three commit sets saved in a tentative transaction: the read set (read object set = ROS), write set (modified object set = MOS), and new set (new object set = NOS). The definition of these sets is simple for a connected FE. The ROS is the set of orefs for persistent objects read by the transaction. The MOS is the set of orefs and new fields of persistent objects that were written by the transaction and the NOS is the set of orefs and fields of new objects created by the transaction that are reachable by some persistent object.

The definitions of these commit sets change for tentative transactions. In a tentative transaction the ROS may consist of both persistent objects and objects that are *tentatively persistent*. An object is *tentatively persistent* if it was created by some previous tentative transaction in the log that was tentatively committed but not yet committed at the OR and is logged. This also applies to the MOS in a tentative transaction - it can have both persistent and tentatively persistent objects.

Objects are tentatively persistent since for a given tentative transaction, TT_k where 1 < k ≤ n, ROS_k may contain objects from NOS_l where 1 ≤ l < k. To TT_k the state of the objects at the FE cache should look as if TT_l had succeeded and therefore any objects from NOS_l should be considered persistent. But since the objects from NOS_l have not actually been committed at the OR, they are considered tentatively persistent. This difference affects the logging of objects which is done on-the-fly as application code reads, modifies, or creates objects.

Figure 3-1, shows examples of ROS, MOS, and NOS sets for tentative transactions. In TT_2, the object g is in ROS_2 since it is tentatively persistent from TT_1. Similarly in TT_{n-1}, both ROS_{n-1} and MOS_{n-1} contain object e which is tentatively persistent from TT_1.

Translating the transaction log into a tentative transaction is not completely straightforward. When a newly created object is logged in the transaction log, it is not yet known whether that object is reachable by some persistent object. So the newly created objects logged in the transaction log may contain objects not reachable by some persistent object. Since the NOS in a transaction is defined as the set of orefs and fields of new objects created by the transaction that are *reachable by some persistent object (or tentatively persistent)* , the NOS can not simply be the same set of objects as the newly created objects in the transaction log. The reason for only saving objects reachable by some persistent or tentatively persistent object in the NOS is that it minimizes the space needed in the log and it also minimizes the number of objects sent to the OR in a commit request. If objects not reachable by any persistent object are committed at the OR, they would eventually be garbage collected at some future time. In addition, objects not reachable by a persistent object can be discarded at the FE to save space after a tentative commit as long as the application did not have any handle on them.

Therefore, when an application tentatively commits, it is necessary to create the NOS for the tentative transaction by traversing the references of the MOS. By iterating through each of the objects in the MOS, references to newly created (or non-persistent) objects can be identified and placed into the NOS. In addition, the references of objects added to the NOS must also be traversed since they may also potentially have references to newly created objects. It is sufficient to only traverse the MOS to create the NOS since an object will be included in the NOS only if it is referenced by a persistent object. Since the objects were newly created, some persistent object must have been modified in order to reference the new object. Therefore an object will only be reachable from some persistent object if it is reachable by some reference in an object in the MOS.

Commit sets store objects using only orefs. An oref is an id for an object by which the OR refers to the object. It identifies an object including the page and segment in which it resides in the object store. An object may have references to other objects in its fields. These references can either be in the form of ROT entries (pointers to local memory as described in Section 2.3.2 or in the form of orefs. But commit sets sent between the FE and OR must have objects with references only in the form of orefs since pointers to local memory will not make sense at the OR. The process of converting orefs to local memory pointers is known as *swizzling* and reversing the process is known as *unswizzling* [4]. In a tentative transaction, commit sets must have all of their references to objects unswizzled before they are sent to the OR since the OR will not understand local pointers from the FE.

Unswizzling of objects takes place when a transaction is tentatively committed and the tentative transaction is created from the transaction log. In order to store objects in the commit sets unswizzled, it is necessary to assign orefs to newly created objects since they are not yet known to the OR and do not yet have orefs. Unswizzling references is especially important in disconnected operation since unswizzling a reference allows for the entry in the ROT corresponding to that entry to be freed (by decreasing the references to that ROT entry to 0). Free entries in the ROT are needed for allocating new objects and the FE could potentially run out of free ROT entries if none are ever freed during disconnected operation. Freeing ROT entries occupied by references to persistent objects can help to prevent this. A downside to unswizzling references on a tentative commit is that it will slightly degrade performance of the next transaction if the next transaction accesses objects that were just unswizzled since those objects will then be swizzled again which does incur some overhead.

Alternatively, object references could be unswizzled right before the commit sets are sent to the OR during a reconnect. Then references would be stored as swizzled in the tentative transaction log. However this would cause potentially useful ROT entries to be occupied and also would make moving ROT entries hard.

As mentioned, the NOS set can not be directly created from the transaction log since only newly created objects reachable from a persistent object should be included in the NOS. This process can also be used as a point for unswizzling object references since all of the references are traversed at this point.

One complication of unswizzling references to objects is that since newly created objects are assigned temporary orefs, these temporary orefs must be used throughout the tentative transaction log. So for example, TT_1 creates the object a, assigns it a temporary oref of 10274 and TT_2 reads object a. Since temporary orefs do not encode any sort of location as orefs do (because newly allocated objects are allocated from a heap of free memory and not inside an organized arrangement of pages), object a will be referenced as a location in memory at the FE before TT_2 is created. When TT_2 is stored into the tentative transaction log, references to a must be updated to the correct temporary oref assigned to it in TT_1 (10274). It is necessary to update references to the object because once the tentative transaction log is replayed, if TT_1 commits, the state of TT_2 should be as if TT_1 had committed when he tentative transaction was stored.

In order to be able to handle the abort of a tentative transaction, each tentative transaction in the log must also save the state of each object in the MOS prior to its first modification. Objects in the MOS may be modified multiple times but only the initial state of the object before any modifications is saved in the log and only the state after the final modification in the duration of that transaction is saved in the MOS. An alternative to saving the state of each object in the MOS prior to its modification is to save only the state of a modified object if it has not already been in the MOS of some previous transaction in the tentative transaction log. For example, if tentative transaction, TT_k has $MOS_k = \{a, b, c\}$ and objects a and b were modified in TT_{k-1} but object c had never been modified by any tentative transaction in the log, TT_1 .. TT_{k-1}, then only a copy of object c will be saved with TT_k. The reasoning behind this option is that in order to save as much space as possible, it is not necessary to keep saved states of modified objects if these states can be retrieved from previous tentative transactions in the log. However for this project, space was traded for time because in the case of an abort, it is convenient to have the old states of the modified objects on hand rather than having to search through the log for the states needed to handle the abort.

3.1.3 Reconnect

When the FE reconnects with the OR, synchronization of the log occurs before the FE can proceed with any connected operations. Synchronization with the OR consists of replaying each tentative transaction in the order in which they were committed while disconnected, handling any aborts, and also handling invalidations.

To replay a tentative transaction, the FE will send to the OR a commit request containing the commit sets stored for that tentative transaction in the tentative transaction log. However at the time of sending a commit request to the OR, it is necessary to update temporary orefs to permanent orefs. Permanent orefs are assigned by either checking for free space in the current pages at the FE or contacting the OR for a page with space for new objects (and thus with new orefs) if there is no space at the FE. Since acquiring new orefs requires communication with the OR, while disconnected it is necessary to assign temporary orefs to newly created objects instead. However it could be argued that newly created objects should get permanent orefs when disconnected if they can (by using the space on the pages at the FE) and then only get temporary orefs if there is no space. But this would complicate the replay process since only some new objects would need new orefs and some would already have new orefs. So similar to temporary orefs, for an object that is assigned a permanent oref, it is necessary to update references to that object in subsequent tentative transactions (assuming that the tentative transaction committing that object does commit) in the log before their commit sets are also sent to the OR. Once the commit sets have been updated with permanent orefs for newly created objects, the sets are sent to the OR in a commit request.

The OR will then check if the commit request should be committed or aborted. The request will abort if an object in the read or write set of the transaction has been modified by another FE. The OR will send a message back to the FE indicating whether or not the transaction was committed. When the FE receives the OR's response to the commit request it will either have to deal with a commit or an abort. On a commit the FE must install newly created objects from the tentative transaction into its persistent cache. On an abort, the FE must revert the state of the FE back to its original state before the tentative transaction and delete newly created objects from the volatile heap. To revert the state of the FE back to its original state before a tentative transaction, saved copies of modified objects per tentative transaction in the log are used to change modified objects back to their previous state.

When a tentative transaction is committed, it is handled as any normal connected commit. The objects in the NOS of the commit request must be installed into the persistent cache of the FE. Since the newly created objects were assigned permanent orefs prior to sending the commit request, the assignment of permanent orefs also allocated space in the persistent cache for the new objects. The formerly volatile objects must be copied from the volatile heap to the appropriate pages in the persistent cache.

When a tentative transaction is aborted, it is also handled similarly to normal connected aborts. Space allocated in the persistent cache for objects in the NOS must be freed and volatile objects should be cleared from the volatile heap. In addition, the state of any modified objects from the tentative transaction must be reverted back to their state before the tentative transaction by using the saved information in the tentative transaction log. But for a tentative transaction, in addition to reverting modified objects to their state before the tentative transaction, subsequent tentative transactions that depend on that tentative transaction must also be aborted. Tentative transaction TT_l depends on TT_k where $k > l$ if:

$$(ROS_k \cap MOS_l \neq \emptyset) \vee (MOS_k \cap MOS_l \neq \emptyset) \vee (ROS_k \cap NOS_l \neq \emptyset) \vee (MOS_k \cap NOS_l \neq \emptyset).$$

This defines dependency since TT_k can not be committed if it read or modified objects that were in an invalid state (as indicated by the abort of TT_l). Waiting for TT_k to be aborted by the OR is not desirable since if TT_k contains objects from NOS_l in its commit sets then the OR will not be aware of these objects and it is better to simply remove TT_k from the log before it is sent to the OR in the synchronization process.

In checking for dependencies, if a subsequent tentative transaction in the log aborts due to its dependency on an aborted transaction, then any tentative transactions dependent on it must also abort. This means that dependency checks will scan the entire log and abort any transactions that are directly or indirectly dependent on the aborted transaction. One tricky part of this is the order in which the state of objects in a tentative transaction's MOS are reverted to their saved state or *undone*. For example, TT_k with MOS_k = {a} aborts. TT_l with MOS_l = {b} is dependent on TT_k because ROS_l = {a,b}. TT_m with MOS_m = {b} is dependent on TT_l because ROS_m = {b}. So here is a chain of dependencies and after all of the dependent aborts have been processed, the state of object a should be as it was before TT_k and the state of object b should be as it was before TT_l. In the case of object b, it is important that its state be undone backwards, first to the saved state in TT_m and then to the state saved in TT_l. Therefore when aborts are processed, the dependencies are found in a forward scan but undoing the state of each is done in a backwards process through each of the dependent tentative transactions.

After the FE processes an abort by reverting the state of modified objects and deallocating space in the persistent cache, it must notify the application of the failure. It does this by simply returning to the application a set of tentative transaction id's for each tentative transaction that aborted. This is done after all tentative transactions in the log have been tried since upon notification of an abort, an application may wish to perform conflict resolution by committing another transaction which would complicate the replay of the log. The application can match these id's with id's returned by the FE on tentative commits. These id's can then be matched with any context that the application may have saved. The usage of this information to help in conflict resolution is discussed in section 3.2.3 on application level conflict resolution.

Once the entire log of tentative transactions has been processed by sending each of the tentative transactions to the OR in a commit request and then processing the response, invalidations must be dealt with. In the process of replaying the tentative transactions, the FE may receive invalidation messages containing orefs of objects that have become stale (since these messages are piggybacked with commit request responses). These stale objects may or may not have caused some of the tentative transactions in the log to abort but in any case, those objects must be marked invalid in the FE ROT and persistent cache. In addition, these messages can not be processed as they are received since they may invalidate objects used in the commit sets of a tentative transactions that have not yet been sent to the OR. For example, if TT_2 modified object a, but an invalidation message for object a arrives with the response to TT_1 and the FE acknowledges that it has seen the invalidation of a, then the commit request for TT_2 will succeed (assuming that no other object in its read or write set is invalidated in the time between the acknowledgement and the OR receiving the request). This could potentially cause an unexpected state for object a unless, in the processing of the invalidation message, the FE also aborts any tentative transactions that read or write the objects in the invalidated set of objects. While this is a possibility, processing the invalidations after the entire log has been replayed simplifies the implementation and still maintains the consistency of the objects since in cases such as the example given, the OR will abort any tentative transaction commit request that contains invalidated objects in their read and write sets (as long as the invalidation message has not been acknowledged).

To process an invalidation, the FE must mark the object in the ROT and persistent cache as invalid. When an object is invalidated, any access to that object from the application will cause the object to be refetched from the OR to get the current version of the objects. Additionally invalidations are processed before the application receives the list of failed tentative transaction id's. The reason for this is so that any resolution of the failures will have the correct state of objects.

Figure 3-2 depicts the reconnect process for a sample scenario. The tentative transaction log in this case contains three tentative transactions. TT_1 is aborted since some other FE made a modification to object a which this FE has not yet seen. Object a is in ROS_1 and MOS_1 so the OR must abort TT_1. In addition the OR also send an invalidation message for object a piggybacked with the abort. Since $MOS_2 \cap NOS_1 \neq \emptyset$, the FE will automatically abort TT_2 without even sending a commit request to the OR for it. TT_3 is committed successfully. Then the FE processes the invalidation message and sends the acknowledgement to the OR. Finally the FE passes back to the application the list of tentative transaction id's that failed to commit.

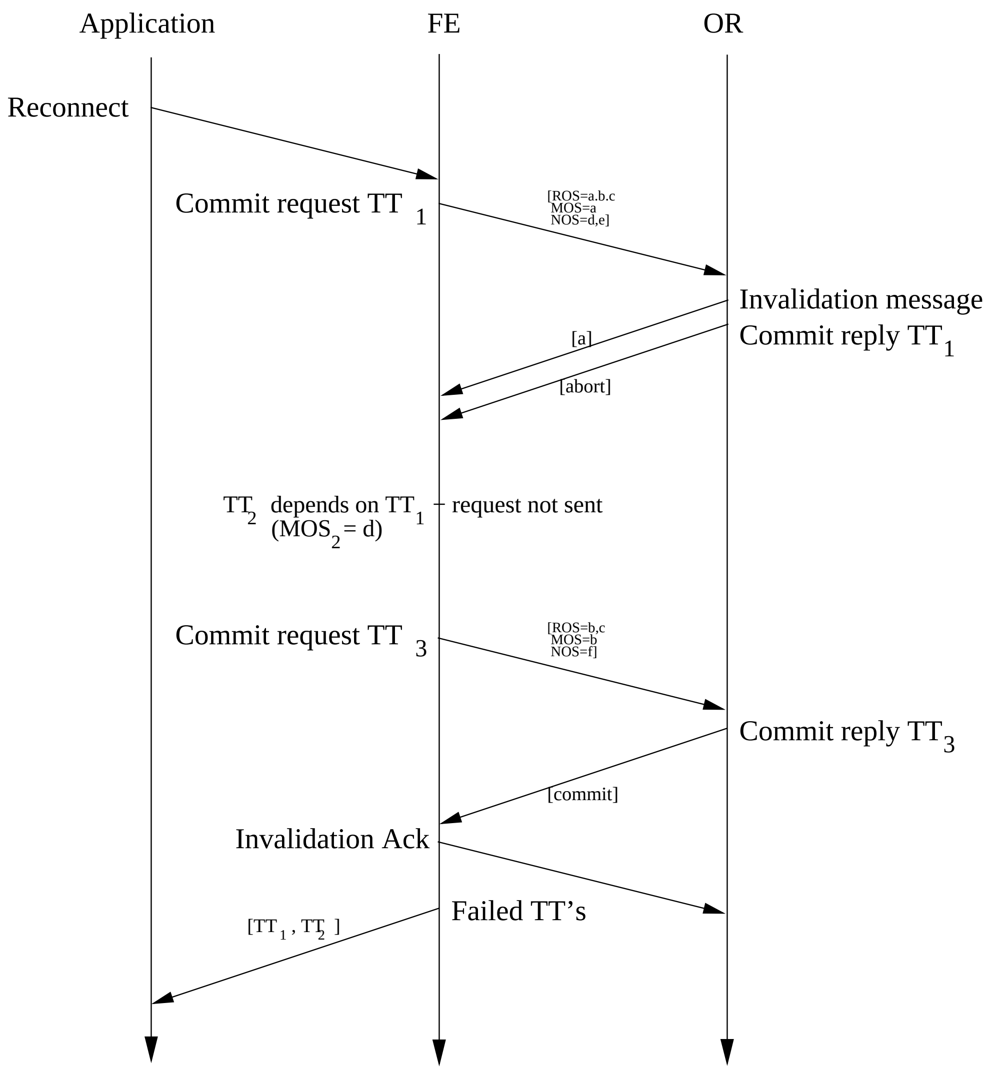

Figure 3-2: Each tentative transaction is replayed in a commit request to the OR in the order in which they were tentatively committed while disconnected. The OR responds with either a commit or abort and invalidations could possibly be piggybacked at the same time.

3.2 Application Support for Disconnected Operation

In addition to the support provided by the FE for disconnected operation, the application must also provide support for disconnected operation, namely support specific to application semantics. This support can be divided into the three components of prior to disconnection, operating disconnected, and reconnecting.

3.2.1 Preparing for Disconnect

Application specified *hoarding queries* are used to prepare the client for disconnection from the OR. A hoarding query is an operation on the persistent objects in the database that causes objects to be fetched or hoarded from the OR into the FE cache prior to disconnecting from the OR. The hoarding query should try to bring objects into the cache that will be of interest to the user while running the application disconnected. Ideally, the user will never request an object that does not exist in the cache while disconnected. Therefore what a hoarding query brings into the cache can really only be determined by the application and perhaps the user.

There are several issues with hoarding queries: what should be hoarded, where should the queries be defined, who should call them, and when should they be called? One possible way to design a hoarding query is to have each class in the schema for an application specify its own hoarding query which reads each of its fields (thus causing those objects to be fetched into the cache if they are not already there). Then starting with the root object of an application, prior to disconnection, the application can call the hoarding query of the application root which will recursively traverse the entire reference graph of the persistent objects for that application. This provides good spatial locality of references because if an application accesses an object it is likely that it will also access that object's fields. However this may not be ideal since it could potentially fetch unwanted objects from the OR since objects are fetched inside pages of objects. For large applications, the user may also not want all objects for a particular application but only a subset of them. For example, in a calendar application the user may want only the objects in his calendar and not all of the objects in all of the calendars in the database. Therefore hoarding queries may be specified at different granularities: at the root of the application, the root for a particular user, or just one object. The difficulty with this then becomes how to allow the user to select one of these options in an understandable manner. However this is beyond the scope of this project. But it is possible to specify hoarding queries in any of these manners.

An application in Thor can prepare for eventual disconnection by always executing a hoarding query on startup. This way if disconnection is unexpected, the application will at least have some set of data to work with while disconnected. However, in the current implementation, disconnecting and reconnecting are application specified events.

3.2.2 Operating Disconnected

When disconnected, requesting to commit a transaction returns a tentative commit with an id to identify that tentative transaction. An application will use this id to identify the tentative transaction if it should abort. The application will associate with this id, context of the operation associated with the tentative transaction. This context includes:

- operation type
- parameters
- priority

Each operation type also has an associated resolution function which attempts to use the saved parameters from the tentative transaction context to resolve the a failure to commit. Whether or not the resolution function is called is based on the priority of the particular operation. The priority will tell the application how many times an operation should be retried before the user is notified and consulted for manual resolution. A priority can be set in a number of ways. The user can be the sole point for determining a priority or the FE can also be involved in determining priority by counting the number of dependent transactions that a transaction has. So the greater the number of dependent transactions, the greater the priority (and retries).

This extra support is necessary since Thor only provides a notification of conflicts and does no resolution. In addition the FE is not aware of type or any other high-level context information once the commit sets are created - it is just a set of objects with fields and orefs. In order to make any sense of an abort, the application must save context to help with this.

3.2.3 Reconnect

On reconnect, as each tentative transaction is replayed to the OR in a commit request, the response received may result in an abort. Once the entire tentative transaction log has been replayed, the application will receive a list of these aborted transactions and will then have to deal with their resolution. It does this by iterating through the list of failed transactions and calling their resolution functions. In the process of calling a resolution function, it is possible that the transaction will be aborted again and a series of nested calls to resolution functions and aborts may occur. In order to prevent the case where a transaction is repeatedly retried forever, the priority of the transaction is used to limit the number of retries.

To resolve a conflict, the application has the flexibility to do a variety of resolutions since the application had control over where conflicts were detected and also has saved context for each failed transaction. While Thor provides conservative abort semantics that guarantee serializability of operations on shared objects, the fact that retries may work since invalidation processing gives the FE the correct state during a conflict resolution, actually allows applications that do not require Thor's conservative abort semantics to achieve their relaxed semantics.

Chapter 4

Evaluation

This section discusses how well disconnected operation in Thor achieves the goals of consistent shared data and flexible conflict resolution through the development of a sample application on top of the Thor framework. In addition, an analysis of performance is presented to discuss the overhead associated with disconnected operation.

4.1 Sample Client Program: A Shared Calendar

A shared calendar system was implemented as an application using the disconnected Thor framework described in section 3. The calendar system maintains a database of calendars where each calendar is associated with a user but multiple users may modify a single calendar. Concurrent modifications to a single calendar are possible. This is a desirable property in many real-life situations such as corporations where an assistant and his boss may both wish to modify the boss's calendar. In the calendar system implemented on top of Thor, a user can add and delete events to and from a calendar. Each event has an associated day and time.

The essential aspect of the design of the calendar application was the data modelling phase or development of the application's schema. The schema is the organization or structure of the data as represented in the database. An important aspect of the data modelling phase for the calendar application was to consider the effect of concurrency on the data. It is especially important in the case of applications in Thor since Thor detects conflicts at the granularity of an object and therefore the design of the object-oriented schema will directly impact what conflicts are detected. In the calendar application, concurrent additions of events to a user's calendar are permissible so long as they do not overlap in time. The correct behavior is for a conflict to be detected only when concurrent updates to the calendar modify events that conflict in time. However, these conflicts are not always significant and in some situations, a user may want more relaxed semantics. These situations can be accommodated with the flexible resolution of conflicts.

For the calendar application, the design process showed just how important the design of the schema is in order to get the correct application conflict semantics. The first iteration of the calendar design maintained a set of event objects and when a user added an event, this event was simply added to an array of events in the calendar object. This is depicted in Figure 4-1(a). However, by designing the calendar in this way, conflicts would be detected on the entire calendar object. For example, if one disconnected client modified the calendar by adding an event and while that client was disconnected, another client also added an event to the calendar, then these two updates would conflict since they both modified the calendar object (the number of events would have changed in addition to the events array). While this behavior was correct in terms of maintaining a consistent set of data, since operations on the calendar would be serializable, it did not capture the correct calendar semantics for a conflict. In addition, the granularity at which conflicts were being detected caused more conflicts to be detected than necessary since any concurrent modifications to the calendar would be considered a conflict.

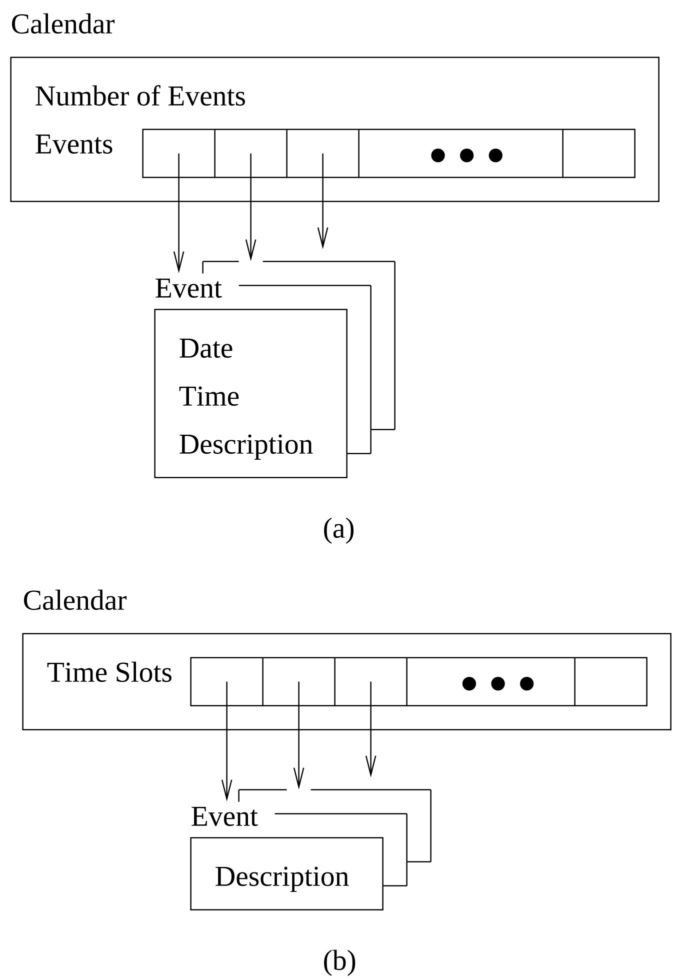

Figure 4-1: (a) Initial calendar schema design that caused conflicts to be detected on any concurrent modifications of the calendar. (b) Revised calendar schema design that causes conflicts to occur on time slots according to correct conflict semantics for the calendar application.

To achieve the correct calendar conflict semantics in the calendar application, the schema was redesigned to detect conflicts at the granularity of time slot objects rather than the entire calendar object. This is depicted in Figure 4-1(b) where the calendar is a set of time slot objects and each time slot can be assigned to some event. With this design, if two users concurrently modify the same time slot object, then a conflict will be detected by Thor. This is the correct semantics for a conflict in a calendar, namely when two appointments are made for overlapping times. However since the user may want to allow this at times, it is important to consider the conflict resolution and the different properties that a user might want to be able to have in his calendar. This will be discussed later in the section.

Both of the designs for the calendar showed how consistency is maintained across multiple clients even when clients disconnect. Tentative transactions save the calendar operations while the users are disconnected and Thor's FE replays these transactions upon reconnection. This method gives fewer guarantees to a disconnected user but still maintains consistent data at the Thor OR. For example, if one user caches a copy of the calendar and another user caches an identical copy of the calendar both from the OR, and both users modify their cached copy of the calendar while disconnected, then there *may* be a conflict. With the second design of the calendar, we maintain consistency in addition to getting the correct conflict semantics. Consistency is maintained since multiple users can concurrently add events to the calendar without having a conflict as long as a transaction does not read stale objects in the calendar. An additional factor to consider in the calendar application design is that a transaction that writes an event should be careful not to include reads of other time slots. This requires that the application developer be very careful in the organization of commit points in the application. For example, in a transaction that adds an event to the calendar, the user can not also read all of the objects in the calendar. It is okay to read only the slot that should be modified and any other dependent slots, however, it should not read all of the time slots. This would cause the same conflicts to happen as the first design!

Conflict resolution in the calendar application is flexible since it was possible for the application to have control over where conflicts were detected. In the case of the calendar, we know that conflicts are over time slots, so if a conflict occurs, we know it is because another user has already modified that same time slot. It is then up to the application, how to deal with this. In order to be able to deal with a conflict the application needs to understand the context for a transaction. Therefore as discussed in section 3, the calendar application saves a description of the high-level operations made by the transaction and any arguments to that operation.

Using the saved context and having fine-grained control over conflict detection through the design of the application schema, any number of policies could be implemented to resolve conflicts. In the case of the calendar some of the possible resolutions are:

- assign tags to events that determine whether or not it can overlap with another event, if it can then resolve the conflict by allowing the overlapping events in the time slot
- have the user suggest several times when entering an event and in the event of an abort for the first-choice time slot, retry entering the meeting with a second-choice time slot
- assign priorities to events and delete the conflicting event if it has a lower priority than the one that was aborted, then add the aborted event into the calendar
- present the conflicting events to the user and ask the user how to proceed

Clearly there are many possible resolutions. In this list, the first three could be considered *automatic* since they could be implemented without ever having any user intervention during the processing of aborts after a reconnect. However, in the cases in which automatic resolution is not possible, manual resolution can be used by asking the user. And as discussed in Section 3.2.3 the first option actually allows the application to have more relaxed conflict semantics since overlapping events could be accommodated - this may be actually be desirable in some cases. Yet the design of the system also accommodates applications with very strict consistency requirements.

4.2 Performance

Providing support for disconnected operation in Thor comes with an overhead in both space and time. This overhead varies by the following factors:

- number of tentative transactions in the log

- read:write:new object ratio in the commit sets
- contention: number of aborts

The remainder of this section will discuss the overhead of disconnected operation in Thor in detail with a number of experiments that make use of the OO7 benchmark. The details of this benchmark are described in [3]. Briefly, the OO7 benchmark provides a comprehensive test of OODBMS (Object-Oriented Database Management System) performance. It is organized into a library of Composite objects and a tree of objects where the leaves of the tree contain references to randomly selected objects from the library. Each Composite object also contains a graph of Atomic objects which reference one another via Connection objects. The actual benchmark is a series of traversals on the tree and queries that exercise different parts of an OODBMS.

In comparison to connected operation in Thor, the design of disconnected operation in Thor has several differences that affect performance. These differences occur both while operating disconnected and upon reconnection. Experiments were conducted with a single FE and OR both running on the same machine. The FE cache size for all experiments was 24MB. The machine is a 450 Mhz Pentium II with 128MB of RAM running Linux Redhat distribution 6.2. Since most tests are measuring the overhead of disconnected operation which is not associated with network costs, the running the OR and FE on the same machine does not play a significant role in the results.

Each experiment uses an OO7 traversal to compare connected with disconnected operation. Connected operation is nearly identical to Thor before the modifications to support disconnected operation were added. The application operates on objects as usual and then commits them to the OR. During connected operation, the application simply waits for the response to a commit request from the OR. In disconnected operation, a commit really has two parts. The first part is to tentatively commit the transaction while disconnected. This places the transaction in the tentative transaction log. The second part is to reconnect and send the tentative transaction from the log to the OR as a commit request. So when comparing disconnected operation with connected operation, it should be noted that disconnected operation consists of two time measurements - the time to tentatively commit and the time to reconnect - whereas connected operations consists of the total time between an application's request to commit a transaction and the time it receives a response.

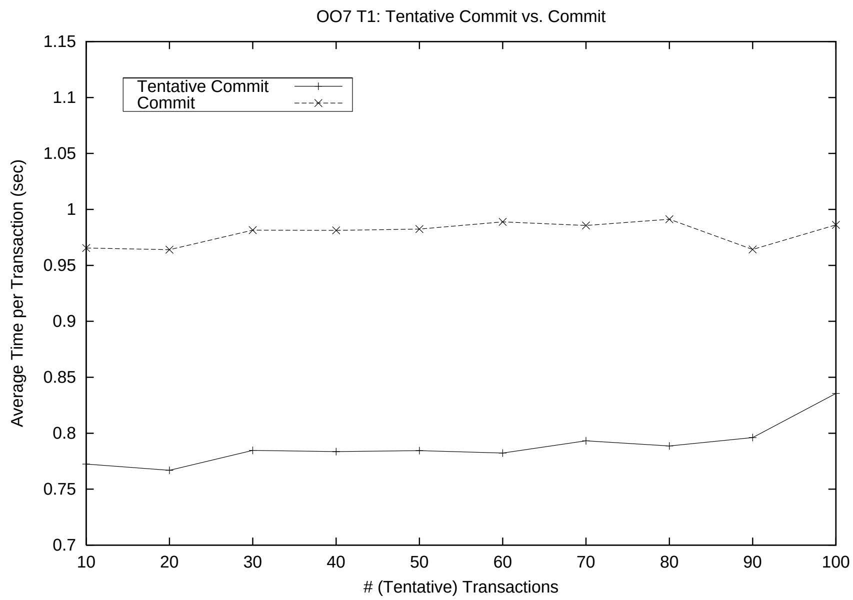

Figure 4-2: Comparison of average time per transaction to commit connected versus time to tentatively commit for 10 to 100 tentative transactions in the log. An OO7 read-only T1 traversal workload was used. Tentatively committing is faster since it does not require communication with the OR.

While disconnected, a tentative commit request results in the unswizzling of the ROS and MOS and the creation of the NOS. This is identical to connected operation. But rather than send the commit sets to the OR as in connected operation, since the FE is disconnected, the commit sets are saved to the tentative transaction log. This means that a tentative commit actually takes less time than a connected commit since there is no communication with the OR. Figure 4-2 depicts the results of an experiment that confirm this. The experiment used the OO7 T1 traversal which is a read-only traversal of the object tree. Figure 4-2 shows that for the workload used, there was on average a 19.3% improvement in the response time for a commit while disconnected versus while connected. This was expected because the commit is only tentative and no communication was made with the OR. The time for a reconnect is discussed in the following section. Additionally, Figure 4-2 shows that the time for a commit in both the connected and disconnected case stays constant with a varying number of transactions. This is correct since the time for a commit is not dependent on the number of transactions in the log.

Upon reconnection, replay of the log incurs two major costs that do not occur in connected commits. However, these overheads occur only if objects are modified or created by transactions while disconnected. To verify this, an experiment using OO7 T1 read-only traversal was used. Figure 4-3 shows the results for the time to commit versus the time to tentatively commit and reconnect for a varying number of transactions between 10 and 100. It shows that the times are relatively the same and both grow linearly with an increasing number of transactions. Interestingly, the time for a tentative commit and reconnection perform slightly better than connected commit. This may be due to a gain in network performance by having a group of messages sent to the OR in succession upon reconnect rather than spread out between transactions while connected. The following will discuss the overhead that occurs when objects are modified or created.

The first overhead incurred is that in preparation to send the tentative transaction as a commit request to the OR, newly created objects that have temporary orefs and these must be updated to be new permanent orefs. Getting permanent orefs is a cost that is also incurred during connected commits, however objects in the MOS and NOS need to be updated with these new orefs. This updating incurs the extra cost of a second traversal of the objects in the MOS and NOS.

While the cost of updating of orefs should only be incurred if new objects are created while disconnected, even when no new objects are created, a cost is incurred if objects are modified. This is due to the implementation of the update procedure that will traverse the MOS of a tentative transaction to check for any temporary orefs even if there were no new objects created while disconnected. This could be fixed by adding a check to each update procedure that checks whether or not there have been any new objects created using the new object table. This was not done because the

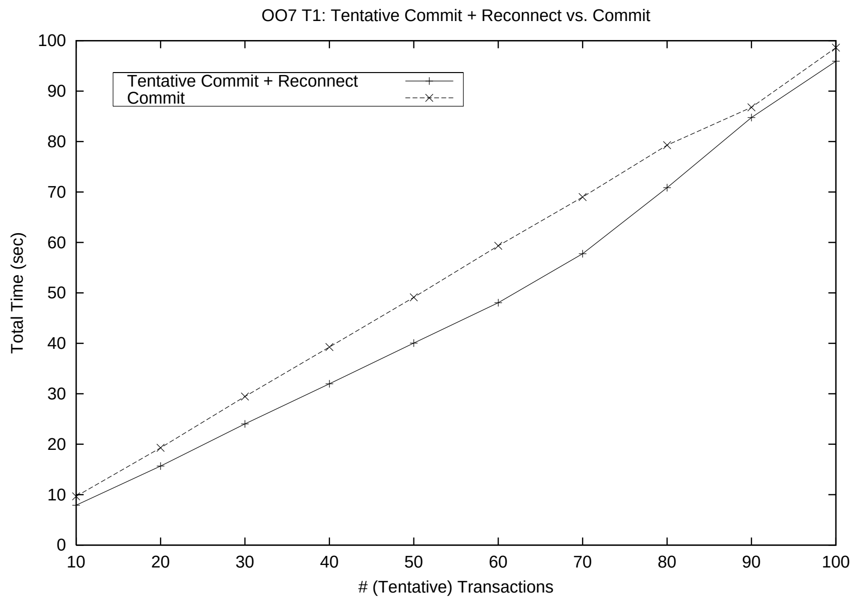

Figure 4-3: Comparison of time to commit connected versus time to tentatively commit and reconnect for 10 to 100 tentative transactions in the log. An OO7 read-only T1 traversal workload was used. There is little or no overhead when only read-only transactions are made while disconnected.

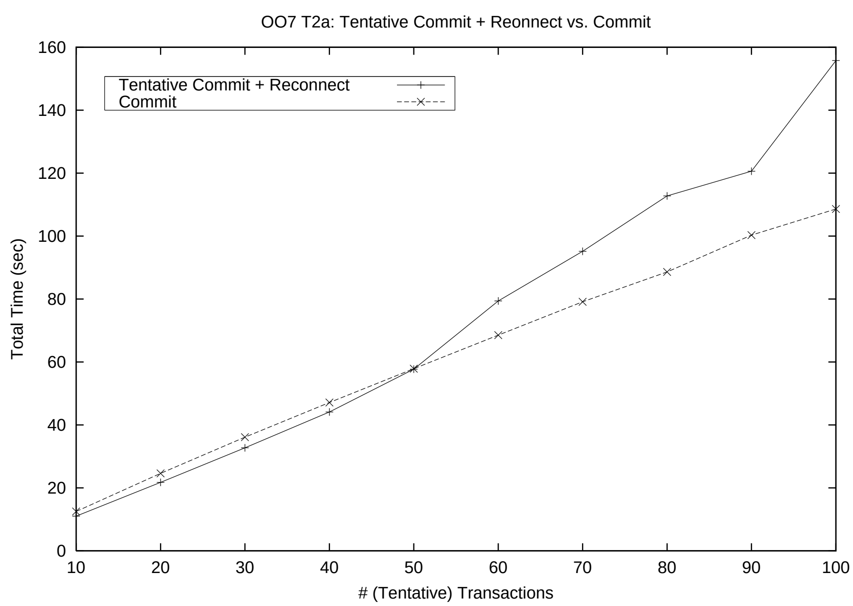

Figure 4-4: Comparison of time to commit connected versus time to tentatively commit and reconnect for 10 to 100 tentative transactions in the log. A read/write OO7 T2a workload was used.

case where no new objects are created while disconnected was believed to be a special case and that it would be most likely that a disconnected transaction would create new objects, in which case the MOS would have to be checked for references to new objects with temporary orefs.

Figure 4-4 shows the difference between the time for a connected commit versus the time to tentatively commit and reconnect a transaction when no new objects are created while disconnected. The workload for this data used the OO7 T2a traversal, which consists of both reads and writes but no new objects giving a 12:1:0 Read:Write:New object ratio in the each transaction's commit The average overhead for disconnected operation in the cases shown in Figure 4-4 is 8.74%. This traversal shows the overhead of reconnecting with varying sized logs that have only modified objects and no new objects. The overhead with new objects is discussed next.

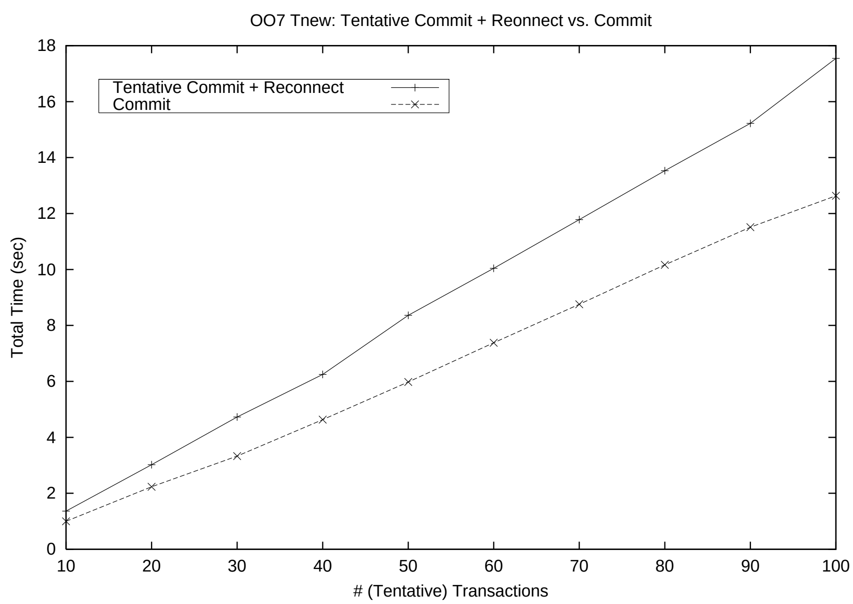

Figure 4-5: Comparison of time to commit connected versus time to tentatively commit and reconnect for 10 to 100 tentative transactions in the log. An OO7 Tinsert workload was used. log.

Figure 4-5 shows the overhead for updating temporary orefs when new objects are created while disconnected. The workload used for this figure was an Insertion query on the OO7 database where 5 new Composite objects are created and then for each new Composite object, 2 references to the object are inserted into randomly selected leaves in the tree of objects (giving a total of 10 new references to Composite objects in the tree). Each transaction consisted of one Insertion query. The average overhead for disconnected operation in the cases shown in Figure 4-5 is 36.32%. The growth of the time to tentatively commit and reconnect is linear with respect to the number of tentative transactions. However it does grow at a faster rate than connected commits.

The second major source of overhead from disconnected operation in Thor comes from aborts. If a transaction is aborted, the log must be updated to abort any dependent transactions. This dependency check has the extra cost of scanning the log with a backwards (as described in Section 3) undo each time an abort happens. On an abort, each subsequent transaction in the log must be scanned to check if that transaction is dependent on the one aborted. Additionally, if any subsequent transactions in the log are aborted, their dependencies must also be checked. This ends up having an exponential time for performing a dependency check scan on an abort. In the worst case, since dependent aborts are removed from the log, if none of the aborts are dependent, then each abort will scan a maximum number of transactions in the log.

Experiments were conducted in both low and moderate contention (abort rate) environments similar to experiments made by Adya for concurrency control studies Thor [1]. The experiments make use of the OO7 T2a traversal rather than the Tnew since the Tnew traversal creates dependent tentative transactions. By using the T2a traversal both the low and moderate abort rate experiments have no dependencies between aborts to show the worst case performance. Figure 4-6 shows the results for an OO7 T2a traversal with a 5% abort rate which is considered low contention. For cases in which 5% of the number of transactions was not an integer, the floor was taken. So for example, a log of 10 tentative transactions has no aborts and a log of 70 tentative transactions has 3 aborts. However, the overhead for disconnected operation with 5% aborts is not clearly exponential compared to connected operation. Figure 4-7 shows the results for an OO7 T2a traversal with a 20% abort rate which is considered moderate contention. The growth for a moderate abort rate is also

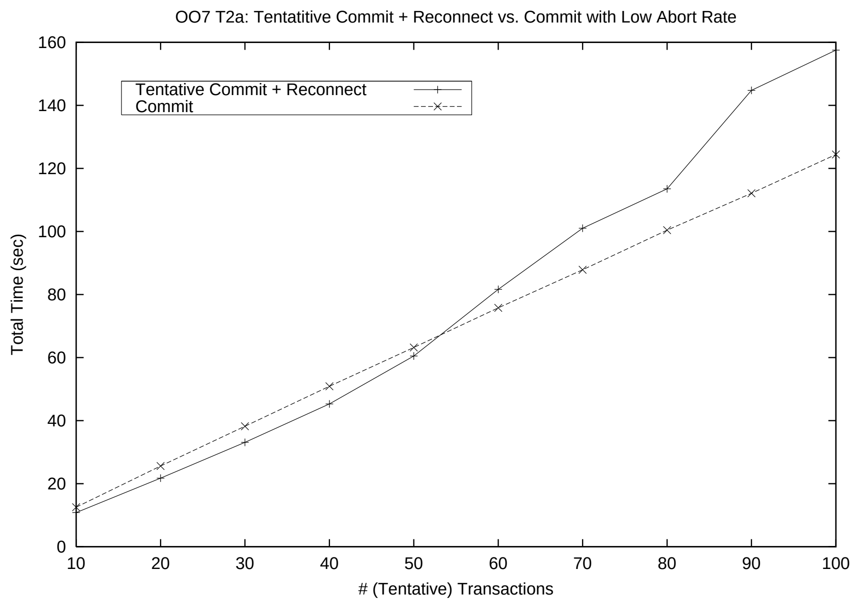

Figure 4-6: Comparison of time to commit connected versus time to tentatively commit and reconnect for 10 to 100 tentative transactions in the log. An OO7 T2a workload was used with a 5% abort rate.

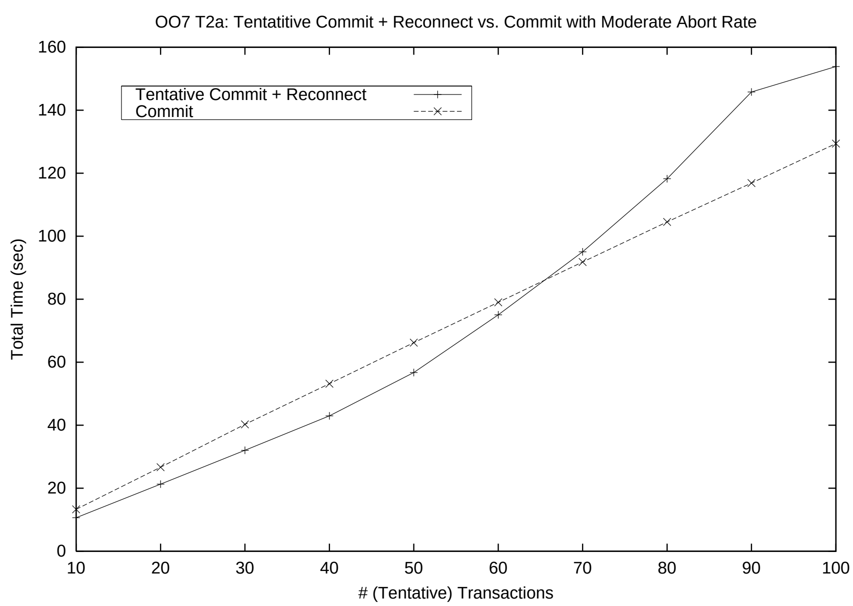

Figure 4-7: Comparison of time to commit connected versus time to tentatively commit and reconnect for 10 to 100 tentative transactions in the log. An OO7 T2a workload was used with a 20% abort rate.

not clearly exponential for disconnected operation. So, to take a closer look at the overhead of aborts on reconnect time, Figure 4-8 shows the total reconnect time for 10 to 100 tentative transactions in the log. By looking at only the reconnect time which includes the processing of aborts, the 5% abort rate is more clearly exponential than in Figure 4-6 since the costs of tentatively committing are not included and the cost of processing aborts during the reconnect become more pronounced. Comparing

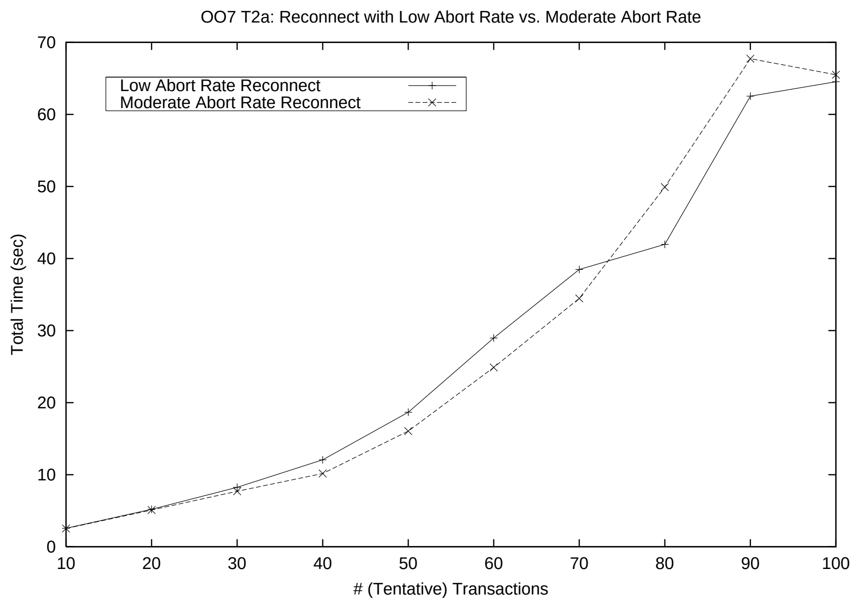

Figure 4-8: Total reconnect time for both low and high abort rates for 10 to 100 tentative transactions in the log. An OO7 T2a workload was used with a 5% abort rate for low contention and a 20% abort rate for high contention.

the reconnect time of the T2a traversal with low contention to the one with moderate contention reveals that their performance is comparable for lower numbers of tentative transactions in the log. Moderate contention has longer reconnect times for 80 to 100 tentative transactions in the log whereas reconnect times with low and moderate contention for 10 to 70 tentative transactions in the log are comparable. The similarity of the two for 10 to 70 tentative transactions in the log is likely due to experimental error since varying network conditions can affect the performance of the reconnect. The time to reconnect for 80 to 100 tentative transactions in the log begins to show longer time for moderate contention over low contention due to the slight improvement of disconnected operation over connected operation for read-only transactions (as seen in Figure 4-3). The improvement for read-only transactions becomes outweighed by the overheads of the writes and aborts.

Referring to Figure 4-4, the total time to commit connected or tentatively commit and reconnect for each of the different sized logs is actually comparable to the results of experiments with aborts. Therefore for the abort rates and number of transactions in these experiments, does not show a high cost for abort scans. However, since Figure 4-8 does indicate and exponential trend in growth with an increasing log size, the cost of abort scan may grow more rapidly for logs with more than 100 tentative transactions.

While the current implementation of abort scans in the design of disconnected operation in Thor indicates an exponential running time, the implementation could be modified to not indicate such a high cost. It is possible to process aborts doing only a single pass over the log by saving information along the way about which objects have been undone by a transaction due to an abort. Dependencies can then be checked against this set of already undone objects in order to abort dependent transactions. A backwards undo is avoided since the set of undone objects is carried along the way. Therefore the cost of abort scans is not inherent to disconnected operation and can be overcome by changes in the implementation.

Chapter 5

Related Work

Providing consistent shared data in the presence of disconnected operation is not a new problem. Many different systems have been implemented that support disconnected operation and the sharing of data. Some of these include Bayou, Rover, and Coda as discussed in Section 2. Having discussed the design of disconnected operation in Thor and evaluated its effectiveness in achieving consistent shared data with flexible conflict resolution, this section revisits each of the systems described in Section 2 to see how they compare.

In general each system uses a similar notion of "tentative" data for data modified while disconnected but has different methods for handling concurrency and conflicts.

Coda supports disconnected operation but it is oriented around a file system. Conflicts are detected only at the granularity of files which gives an application much less control over the semantics of conflicts. Thor on the other hand, can be used for a variety of applications where data easily fits into an object model where objects are small. However if an application is concerned over file-sharing such as in a collaborative document editing system, Coda may actually be a more suitable choice.

Like Bayou, Rover does not provide for any built-in notion of consistency. It is up to the application to define in its procedures checks for conflicts and procedures to resolve them. Thor takes some of the burden of this away from the application by having built-in conflicts detected on objects. While it is true that the application does play a role in defining conflicts since the application schema must be carefully designed to achieve the correct conflict semantics, Thor provides a framework with which the application can work. In addition this framework is a familiar one since it is essentially the framework of an object-oriented programming language.

Bayou and the approach to disconnected operation in Thor are similar in that application-specific conflict detection and resolution are facilitated. Bayou's dependency-check procedure is analogous to schema design in Thor since the manner in which the schema is designed, controls what conflicts are detected. Bayou's merge-proc function is analogous to application conflict resolution in Thor. The difference between the two is that there is no built in notion of consistency in Bayou. While Thor allows for an application to have control over where conflicts will be detected, the serializability of data will not be violated at any point. Thor could perhaps benefit from Bayou's notion of merge-procs. Since Thor applications must now include all conflict resolution code inside the application, it would be beneficial to have in Thor, also a framework for applications to write resolution functions or perhaps even select from a set of common resolution functions. Future work in Thor might investigate the development of this framework.

Chapter 6

Future Work and Conclusions

There is still much research that can be made into disconnected operation in Thor. This work has provided a basis upon which more applications can be built to explore the different schema designs necessary to get correct conflict semantics in a variety of applications. With a variety of applications a set of common features for saving context and resolution procedures may be extracted. Ultimately, this information could be used to develop a middle layer in Thor which offers services to applications for transaction context saving and conflict resolution. For example, if it were important in all applications for arguments to an operation to be saved, it might be useful to add this functionality to the compiler for Thor rather than having each application produce the same code for saving arguments whenever an operation on a persistent object is invoked. Future work on developing more applications could determine whether or not such a layer of application services would be useful in Thor.

In addition, there are many different schemes that can be developed on top of disconnected operation in Thor. One possible scheme which is currently being investigated by Shrira et. al. [19] is to allow clients while disconnected to fetch data from one another. This would allow clients to get up-to-date data and reduce the number of aborts upon reconnection or just get data that they did not cache before disconnecting. Another possible scheme is to investigate being able to commit data while disconnected. If a user who disconnects *owns* a piece of data or knows that no one else will be modifying the same data, a scheme could be devised where that user could have the right to actually commit operations while disconnected, thus bypassing any tentatively committed state. Even further, if a group of users disconnect, they collectively could decide whether or not to commit an operation on a piece of data that only they will modify using some type of voting or quorum scheme.

There are many possible directions for future work with disconnected operation in Thor. This work has provided a different way of looking at the management of shared data in the presence of disconnected operation. It provides a framework in which applications can define conflict detection at the granularity of an object by using the flexibility of object-oriented schema and as a result allows for the application to have enough information to intelligently resolve conflicts. Disconnected operation in Thor suits a variety of applications since it can provide strict consistency rules for applications that require them such as a banking system or airline reservation system. Yet with the framework provided, it also allows applications with more relaxed consistency requirements to have enough control over conflicts and their resolution to achieve more flexible consistency semantics.

Bibliography

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. pages 23–34, 1995.

[2] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. In *ACM Transactions on Database Systems*, pages 163–199, March 1992.

[3] M.J. Carey, D.J. DeWitt, and J.F. Naughton. The oo7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C., May 1993.

[4] M. Castro, A. Adya, and B. Liskov. Lazy reference counting for transactional storage systems. Technical Report MIT/LCS/TM-567, MIT, 1997.

[5] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. Hac: Hybrid adaptive caching for distributed storage systems. In *Symposium on Operating Systems Principles*, pages 102–115, 1997.

[6] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[7] W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asyn-

chronous collaborative applications with bayou. In *Proc. of the Tenth ACM Symposium on User Interface Software and Technology*, pages 119–128, October 1997.

[8] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[9] Irene Greif, Robert Seliger, and William Weihl. A case study of ces: A distributed collaborative editing system implemented in argus. In *IEEE Transactions on Software Engineering*, pages 827–839, September 1992.

[10] Robert Gruber, Frans Kaashoek, Barbara Liskov, and Liuba Shrira. Disconnected operation in the thor object-oriented database system. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[11] Thomas Imielinski and Henry Korth. *Mobility support for sales and inventory applications*, chapter 21, pages 571–594. Kluwer Academic Publishers, 1996.

[12] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, 1997.

[13] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[14] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Usenix Tech. Conf.*, New Orleans LA (USA), 1995.

[15] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628, pages 230–257. Springer-Verlag, New York, NY, 1999.

[16] Q. Lu and M. Satyanaranyanan. Isolation-only transactions for mobile computing. *Operating Systems Review*, 28(2):81–87, 1994.

[17] Evaggelia Pitoura and Bharat Bhargava. Revising transaction concepts for mobile computing. In *Proc. of the First IEEE Workshop on Mobile Computing Systems and Applications*, pages 164–168, Santa Cruz, CA, December 1994.

[18] Evaggelia Pitoura and George Samaras. *Data Management for Mobile Computing*, chapter 3, pages 37–70. Kluwer Academic Publishers, 1998.

[19] Liuba Shrira and Ben Yoder. Trust but check: Mutable objects in untrusted cooperative caches. In *POS/PJW*, pages 29–36, 1998.

[20] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[21] Gary Walborn and Panos Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proc. of the 14th IEEE Symposium on Reliable Distributed Systems*, September 1995.

[22] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. In *ACM SIGPLAN Symposium on Programming Language Issues in Software Systems*, pages 244–269, April 1985.