# An Approach to Disconnected Operation in an Object-Oriented Database

Sidney Chang and Dorothy Curtis
sidchang@alum.mit.edu, dcurtis@lcs.mit.edu
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA

## Abstract

*With the advent of mobile computers, new challenges arise for software designers. This paper focuses on disconnected operation: making mobile computers work well on shared data whether the network is available or not. Initially the shared data is cached on the mobile computer. Modifications and additions to this cached data will be reconciled with the shared data when the mobile computer is reconnected to the network. Conflict resolution will be used to reconcile conflicting changes. In this paper, we examine these issues by adding support for disconnected operation to Thor, an object-oriented database.*

## 1 Introduction

As computers become more mobile, software needs to be adapted to work well, whether a network is available or not. The challenging aspect is to function without a network and still have local changes integrate easily with other data when the network is available. Initially the mobile computer needs to cache relevant data before disconnecting from the network. When network accessibility again becomes available, additions and modifications to the cached data need to be reconciled with the original data. Beyond this if two users have modified the same cached data while disconnected, some form of conflict resolution must be used to integrate these changes. This all must be done without violating serializability.

For example, a travelling salesperson commuting to the office would like to use his hand-held device to enter an appointment into his calendar which is stored in a central database at work. The hand-held device has a cached version of the salesperson's calendar which was downloaded the night before. Meanwhile back at the office, the salesperson's assistant is sitting in front of a desktop machine connected directly to the central database. The assistant enters an appointment into the salesperson's calendar for 2 PM to 4 PM, which is automatically updated in the central database. After the database is updated to reflect the assistant's change, the salesperson on his way to work now has stale data in his calendar. He thinks that he is free from 2 PM to 4 PM when he actually has an appointment for that time. To make things even worse, the salesperson, on his way to work, also enters an appointment into his calendar for 1 PM to 3 PM. This causes a conflict which must be resolved once the salesperson gets to work and reconnects his hand-held device to the database.

Several general properties become apparent from the example described. *Concurrent modifications to the same data may not always be undesirable.* In the case of the calendar, if the assistant had entered an appointment for 12 PM to 2 PM while the salesperson made an appointment for 3 PM to 5 PM, the same shared calendar data would be concurrently modified yet this would not violate the consistency of the data even though the salesperson modified his calendar while it contained stale data. *Conflicts are based on application semantics.* In the calendar, a conflict is an overlapping of appointments in time but could be completely different in another application. *From the user's perspective, automatic resolution of conflicts upon reconnection is desirable but may not always be possible; thus flexibility in resolving conflicts is important.* The salesperson may want all of the entries that he adds to the calendar to take precedence over others so his assistant's conflicting entry would have been deleted to make room for his entry. However there may also be special cases where the salesperson would not want this to apply.

### 1.1 Problem Statement

The problem of concurrency and shared data has been studied at length in the context of databases. The problem of shared data and disconnected operation has also been studied a great deal in the context of network partitions. But the problem of shared data and disconnected operation in the context of mobile devices changes because disconnections are more frequent and more predictable [12]. As a result, conflicts will be more likely. Therefore intelligent conflict resolution is necessary since the user of the mobile device will not want to lose all of the operations that he or she has performed while disconnected.

The problem that this paper addresses is how to build a system that manages shared data in the presence of disconnected operation. This system must address the issues of using stale data while disconnected and dealing with conflicting updates upon reconnection.

### 1.2 Achieving Consistency

To achieve consistency a system can either use a *pessimistic* or an *optimistic* approach. In an optimistic scheme, users are allowed to modify shared data that may be concurrently accessed by other users. If an optimistic scheme is employed and users can be disconnected, some form of conflict resolution is required.

Pessimistic schemes prevent conflicts from occurring by permitting modifications only to that shared data for which the user has a lock. For the mobile computing setting, requiring the possession of locks to modify data limits the availability of data for other users.

Another aspect of achieving consistency with shared data, is maintaining serializability of data. Thus, a common way to handle concurrent accesses to shared data is to use transactions to aid in achieving both consistency and serializability of operations on shared data. Traditional properties of transactions are: atomicity, consistency, isolation, and durability (ACID) [10].

Various transaction models for mobile computing have been studied in [10, 5, 9]. Each is similar in that weaker forms of transactions (i.e. *weak*, *tentative*, or *second-class* transactions) are defined for transactions made on data local to a mobile device while disconnected. Using this weaker notion of transactions or *tentative transactions* allows for a system to have both consistency and an optimistic scheme in the presence of disconnections. While disconnected, tentative transactions operate on locally cached data. Each tentative transaction is logged at the disconnected device. Upon reconnection, each tentative transaction will either commit or abort as it is replayed against the shared data.

### 1.3 Flexible Conflict Resolution

Tentative transactions result in the need for intelligent conflict resolution. Since potentially all of the tentative transactions made while disconnected could be aborted, it is important that the system employ conflict resolution for those aborted transactions so that the disconnected user's operations are not lost.

Dealing with conflicts or aborts that occur upon reconnection is not simple. The problem is that resolution of conflicts is defined by application semantics and providing general support for a variety of applications is hard. Systems can automatically try to resolve conflicts. Another way to resolve the conflicts is to consult the user upon a failure. Conflict resolution can also be left up to the application since it is best aware of its own semantics. In the end, the most complete approach to resolving conflicts is a combination of system, application, and user support

Examples of such systems are Coda, Bayou, and Rover [7, 4, 6]. These systems will be discussed in greater detail in Section 5 and compared with the system presented here.

## 2 Thor Overview

This project uses the Thor distributed object-oriented database system [8]. This paper proves the serializability of Thor and its ACID properties. In this section, an overview of the Thor architecture is presented.

### 2.1 Thor Architecture

Thor provides a persistent store of objects where each object has a unique identity, set of methods on a per type basis, and state. The system has a client/server architecture where servers are repositories of persistent objects. The server or object repository (OR) consists of a root object plus all persistent objects that are reachable from 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 consist of a front end (FE) and an application. The FE handles caching of objects from the OR and transaction processing. Applications operate on cached objects at the FE inside transactions and commit transactions through the FE to the OR.

### 2.2 Objects in Thor

Each object is uniquely identified by an identifier known as an *oref*. Orefs are also used to locate an object within pages at the OR and FE. At the OR's objects are known only by their orefs. 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 used temporarily by the application and that will 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 are stored in the volatile heap and do not have orefs until they are committed and assigned oref's by the OR. To facilitate program access to objects at the FE in the persistent cache, orefs are mapped to local memory addresses.

### 2.3 FE Transaction Logging and Committing Transactions

Applications make high level operations on objects. These high-level operations on objects are reduced to reads, writes, and creations of objects. Each of these is logged by the FE in order to create the correct read, written, and created object sets to be sent to the OR in a commit request.

An application completes a transaction by making a request to the FE to commit the transaction. The FE processes this request by collecting all of the logged 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. 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 read or wrote invalidated objects. The OR maintains a per-FE set of invalidated objects. These are objects whose state has become invalid since the time the FE cached them. An object at an FE is invalidated when another FE successfully commits a transaction modifying that object since the cached version is now stale. 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.

### 2.4 Summary of Thor

Thor provides transaction controlled access to shared data. Its optimistic concurrency scheme is appropriate for disconnected operation and its object-oriented nature should provide some benefits.

## 3 Disconnected Operation in Thor

In the previous section, Thor was introduced. This section describes how we added disconnected operation to Thor [3].

The approach to disconnected operation in Thor is to use tentative transactions to manage shared data while disconnected and to provide a framework that enables the application to handle conflict resolution. The extension of Thor to support disconnected clients has two main aspects: extensions to the application and extensions to the FE (caching and per transaction processing).

### 3.1 FE Support for Disconnected Operation

FE support for disconnected operation can be divided into three phases. The first is preparation for disconnection. The second is operating disconnected: handling transactions differently. The third is reconnecting with the OR: processing the 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 needs to prefetch objects into the cache by processing queries specified by the application. The application may need a special prefetch query to ensure that all the relevant data is cached in the FE. This will be discussed in Section 3.2.1.

#### 3.1.2 Operating Disconnected

Once the client has disconnected from the OR, an application will attempt to commit transactions as it normally would while connected. While disconnected, a commit becomes a *tentative commit* meaning that the commit could potentially be aborted by the OR upon reconnection. While disconnected, applications will operate the same as when they are connected by making operations on objects inside transactions. These operations will change the state of cached objects at the FE.

The FE logs tentative transactions in the tentative transaction log. This log saves enough state per tentative transaction in order to replay each tentative transaction once the FE is reconnected with the OR. The application is given an id for each tentative transaction that the application can associate with operations performed during that transaction. This information can be used later to assist the user in recovering from an abort. Figure 1 depicts an example tentative transaction log.

**Tentative Transaction**

The model used in connected Thor has the FE maintain transaction information on a per transaction basis. This

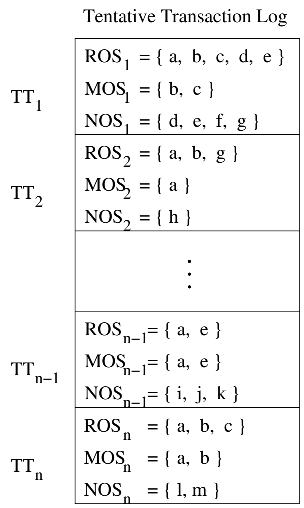


**Figure 1. Tentative Transaction Log**

information, also known as commit sets, consists of the read object set (ROS), the modified object set (MOS), and the new object set (NOS) created during a transaction. When the application commits the transaction, during connected operation, these sets are inserted directly into a commit request to the OR. But, while disconnected, these sets are maintained in the tentative transaction log.

The definitions of the 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 tentative transaction that was tentatively committed but not yet committed at the OR. This also applies to the MOS in a tentative transaction: it can have both persistent and tentatively persistent objects. In Figure 1, $TT_2$, contains object *g* in $ROS_2$ since it is tentatively persistent from $TT_1$.

In a tentative transaction, commit sets must have all of their references to objects in oref format before they are sent to the OR. Temporary orefs are assigned to objects that are created by tentative transactions. In Figure 1, when $TT_2$ is stored into the tentative transaction log, references to *g* must be updated to the correct temporary oref assigned to it in $TT_1$.

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.

#### 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.

Before sending a commit request to the OR, it is necessary to update temporary orefs in the NOS to permanent orefs. Permanent orefs are assigned either from free space in the current pages at the FE or by contacting the OR. In addition, the MOS and NOS may contain temporary orefs for tentatively persistent objects and these must be updated as well. Then the FE sends to the OR a commit request containing the commit sets stored for the tentative transaction.

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. When the FE receives the OR's response to the commit request it will process either 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 uses the saved copies of modified objects to revert them back to their original state before the tentative transaction and then deletes newly created objects from the volatile heap.

When a tentative transaction is aborted, it is handled similarly to a connected abort. 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:

$$(ROS_l \cap MOS_k \neq \emptyset) \vee (MOS_l \cap MOS_k \neq \emptyset) \vee (ROS_l \cap NOS_k \neq \emptyset) \vee (MOS_l \cap NOS_k \neq \emptyset).$$

This defines dependency since $TT_l$ can not be committed if it read or modified objects that were in an invalid state (as indicated by the abort of $TT_k$). Because the abort of $TT_k$ causes the objects in $NOS_k$ to be deleted, overall bookkeeping is simpler if transactions involving references to $NOS_k$ are removed at the same time.

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. It is important to be careful about the order in which the state of objects in a tentative transaction's MOS are reverted to their saved state. 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 entire log of tentative transactions has been processed, invalidations are handled. In the process of replaying the tentative transactions, the FE may receive invalidation messages containing orefs of objects that have become stale. These stale objects must be marked invalid in the persistent cache.

After the FE processes all tentative transactions and invalidations, it must notify the application of any failures. It does this by returning to the application a set of tentative transaction id's containing the id of each tentative transaction that aborted.

Figure 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 $TT_1$. The OR also sends an invalidation message for object $a$. Since $MOS_2 \cap NOS_1 \neq \emptyset$, the FE will automatically abort $TT_2$ without 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.

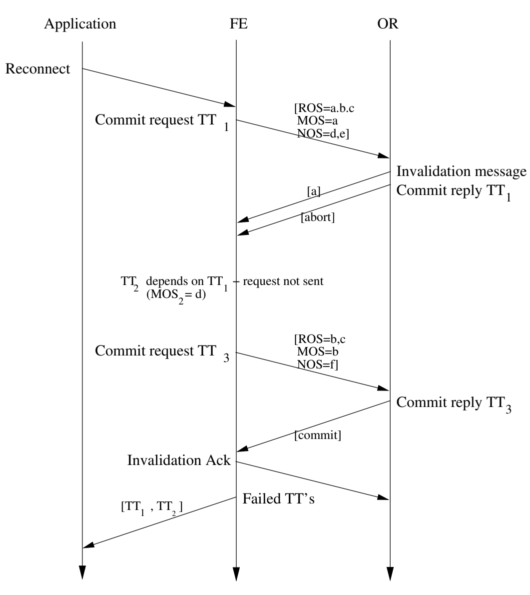


**Figure 2. Synchronizing the Log**

### 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 preparing for disconnection, operating disconnected, and reconnecting. A specific example of an application and the support it provides will be discussed in Section 4.1.

#### 3.2.1 Preparing for Disconnect

Application specific *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.

#### 3.2.2 Operating Disconnected

When disconnected, attempting to commit a transaction returns an id for the tentative commit. An application will use this id to identify data associated with the tentative transaction if it should abort. This contextual data can include operation type, parameters, or priority. Each operation type could also have an associated resolution function which attempts to use the saved parameters from the tentative transaction context to resolve a failure to commit. This extra support is necessary since Thor provides only a notification of conflicts and does no resolution itself.

#### 3.2.3 Reconnect

On reconnect, after the entire tentative transaction log has been replayed, the application receives a list of the id's of aborted transactions and then deals with their resolution. It does this by iterating through the list of failed transactions and calling their appropriate 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.

To resolve a conflict, the application has the flexibility to do a variety of resolutions since the application has control over where conflicts are detected and also has saved context for each transaction. While Thor provides conservative abort semantics that guarantee serializability of operations on shared objects, successful retries of a failed tentative transaction actually allow applications that do not require Thor's conservative abort semantics to achieve more relaxed semantics.

## 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 from 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. 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. In the data modelling phase it is important to consider the effect of concurrency on the data. It is especially important in Thor since conflicts are detected 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.

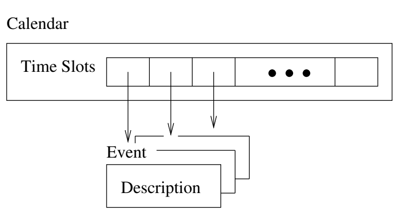


**Figure 3. Calendar Schema**

To achieve the correct calendar conflict semantics in the calendar application, the schema was designed to detect conflicts at the granularity of time slot objects rather than the entire calendar object. This is depicted in Figure 3 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.

With the described 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 application should not also read all of the objects in the calendar, since this will increase the likelihood of an abort.

Conflict resolution in the calendar application is flexible since it is possible for the application to have control over where conflicts are 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 to deal with this conflict. 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 3.2.2, the calendar application saves the high-level operations made by the transaction and any arguments to the operations.

Using the saved context and having fine-grained control over conflict detection through the design of the application schema, any number of policies can be implemented to resolve conflicts.

### 4.2 Performance

The overall performance of the Thor system is discussed in [1]. Thor compares favorably with similar systems in terms of throughput and scalability. This section discusses the added overhead of supporting disconnected operation. First, the number of tentative transactions is limited by the amount of memory in the client. The overhead varies by the number of tentative transactions in the log, the read:write:new object ratio in the commit sets, and the level of contention or percentage of aborts for a given number of tentative transactions in the log. The remainder of this section will discuss the overhead of disconnected operation in Thor using experiments based on the OO7 benchmark which provides a comprehensive test of object-oriented database management system performance. The details of this benchmark are described in [2].

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. The OR was run on a 400 Mhz dual Pentium II with 128 MB of RAM running the Linux Redhat distribution 6.2. The FE was run a 450 Mhz Pentium II with 128 MB of RAM running Linux Redhat distribution 6.2. The FE cache size for all experiments was 24 MB. All communication between the two machines was on an isolated network so that variations in network traffic would not affect the experimental results.

Each experiment uses an OO7 traversal to compare connected with disconnected operation. The difference between disconnected and connected operation occurs at commit points. During connected operation, the application simply waits for the response to a commit request from the OR. In disconnected operation, a commit 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.

Upon reconnection, replay of the log incurs two major costs that do not occur in connected commits. The first overhead incurred is, in preparation to send the tentative transaction as a commit request to the OR, newly created objects that have temporary orefs must be updated to have 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 for all tentatively committed transactions in the log.

We evaluated the average overhead to be 36.32% for updating temporary orefs on logs ranging from 10-100 tentative transactions with a workload of an OO7 insertion query with 5 new composite objects and 2 modified objects per transaction. 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. This is due to an increasing number of permanent orefs to search through when replacing temporary orefs with permanent ones.

The second major source of overhead from disconnected operation 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 undo (as described in Section 3.1.3) each time an abort happens.

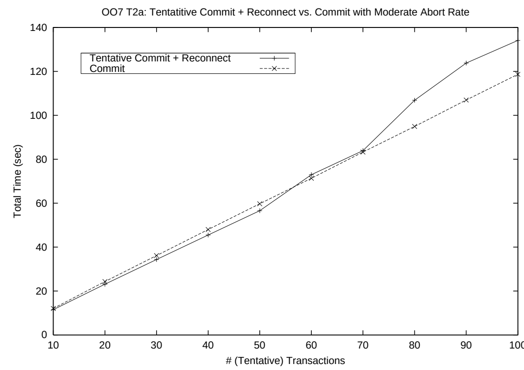


**Figure 4. Abort Overhead (20% abort rate)**

Experiments were conducted in both low and moderate contention (abort rate) environments similar to experiments made by Adya for concurrency control studies in 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 transactions to show the case of maximal scanning. Figure 4 shows the results for an OO7 T2a traversal with a 20% abort rate which is considered moderate contention. The results for a low 5% abort rate show similar trends.

## 5 Related Work

Providing consistent shared data in the presence of disconnected operation is not a new problem. Researchers have analyzed the issues and systems have been implemented that support disconnected operation and the sharing of data.

The major relevant analytical work is [11]. Disconnected operation in Thor largely implements the behavior described in the caching example in this paper: the mobile computer performs weak transactions while disconnected. These transactions are committed only if they do not conflict with the strict transactions at the server. Beyond actually implementing this model, we have started to understand how applications can make use of this system.

Some implementations include Bayou, Rover, and Coda as mentioned in Section 1.3. Having discussed the design of disconnected operation in Thor and evaluated its effectiveness in achieving consistent shared data with flexible conflict resolution, this section visits each of the systems described in Section 1.3 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.

Both Bayou and Rover do not provide for any built-in notion of consistency. It is up to the application to define in its procedures, checks for conflicts and the 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 add to Thor, a framework for applications to write resolution functions or perhaps even select from a set of common resolution functions.

## 6 Conclusion

This paper has described a system that can, with more experimentation, be extended to support a variety of applications. These applications will behave well using shared data whether the network is available or not.

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.

## Acknowledgements

The authors are grateful for support from members of the MIT Project Oxygen partnership: Acer, Delta, Hewlett Packard, NTT, Nokia, and Philips and from DARPA through the Office of Naval Research contract number N66001-99-2-891702. We also wish to thank Chandra Boyapati, Liuba Shrira, Hong Tian, John Ankcorn, Steve Garland, Paul Kim, and Hari Balakrishnan for their thoughtful suggestions.

## References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, US, 1995.

[2] M. Carey, D. DeWitt, and J. 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.

[3] S. Chang. Adapting an object-oriented database for disconnected operation. Master's thesis, Massachusetts Institute of Technology, 2001.

[4] 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.

[5] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, 1996.

[6] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, 1997.

[7] 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.

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

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

[10] E. Pitoura and B. 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, Dec. 1994.

[11] E. Pitoura and B. Bhargava. Data consistency in intermittently connected distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):896–915, 1999.

[12] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*, chapter 3, pages 37–70. Kluwer Academic Publishers, 1998.