MX: Mobile Object Exchange for Collaborative Applications*

Liuba Shrira and Hong Tian

Department of Computer Science
Brandeis University
Waltham, Massachusetts 02454, USA
liuba@brandeis.edu hongtian@cs.brandeis.edu

Abstract. MX is a new mobile caching system for collaborative applications accessing data residing in large storage repositories. MX supports *mobile exchange* – a direct user-to-user object transfer. Mobile exchange (MX) makes mobile computing more effective because it enables certain kinds of collaborative work that would be impossible otherwise. MX allows disconnected peers to learn of recent unknown updates, and to apply these updates to locally-cached data. MX validates the exchange, merging the more recent and modified data. MX combines efficient support for coarse-grained data transfer with efficient fine-grained validation, in a way that avoid the problem of false sharing. Performance evaluation of the MX prototype indicates that for transactional applications, the extra cost required to support mobile exchange is moderate. Moreover, the extra cost is offset by the cost of accessing remote repositories over high-latency networks.

1 Introduction

As mobile computing becomes more pervasive, users expect levels of data consistency and integrity that are not supported by current systems.

Disconnected access to consistent shared files is by now a well-understood problem [29]. Mobile users manipulate locally-cached copies of files and periodically validate their changes against the "master copy" of the file stored at the server. Validation encompasses two problems: first, detecting conflicts, which usually is done with timestamps on files, and second, reconciliation of conflicts which is usually done in an application specific way (sometimes by just asking the user).

Disconnected access becomes a harder problem if the shared persistent data is not files but instead consists of many small component objects, e.g. molecular structures or construction designs manipulated by CAD applications. Since the component objects can be small, much smaller than files, its is not feasible to associate a timestamp with each small modified object. Putting timestamp on

* This research is supported in part by the National Science Foundation (CCR-9901699).

coarser grain objects, such as physical memory pages, creates the problem of *false sharing* when accesses to unrelated objects appear to conflict because they occur on the same physical page. Fine-grained validation techniques that avoid the problem of false sharing in disconnected systems are subject of ongoing research [9, 10, 25, 43].

The contribution of this paper is to extend the validation and reconciliation techniques to systems that support *mobile exchange*, the ability of a user to acquire missing and more recent versions of shared data directly from another user while disconnected from the master copy. Mobile exchange makes mobile computing more useful because it allows to accomplish collaborative work that would be impossible otherwise.

Consider a design team from a large engineering firm visiting a client company. The team is preparing a presentation using the design documents for an initial proposal. The design documents are cached on laptops, while the master copies are stored at the firm's home repositories. Part-way through the visit, the team needs to revise the proposal, incorporating design elements from a laptop of another team member. The team needs to change the design and merge the changes in a safe and coherent way. The modifications need eventually to be propagated to the home repository. The problem is the team does not have connectivity to the repository servers.

Direct data exchange would allow the team to prepare a revised proposal using the laptop copies of the data while disconnected. Fine-grained validation and reconciliation would avoid false conflicts upon reconnection, and multi-object atomicity would ensure that all the modifications are installed atomically, avoiding the potential danger of creating an inconsistent set of data objects in the presence of conflicts and independent reconciliation actions [24].

No mobile storage system today provides all these features. Peer-to-peer replication systems that support direct data exchange either use "update anywhere" replication that allows replicas to diverge [24], or require (for validation) a complete replica of the storage system to be available at each peer, ruling out large systems [27, 32, 42]. Master-copy systems, such as databases [24, 25], or file systems [38], that do not require full-replication, provide no support for direct data exchange between users.

MX is a new mobile caching system for collaborative applications accessing large-scale object storage systems. It allows mobile users to exchange complex data in an efficient way, supports fine-grained validation and coarse-grained reconciliation, and insures that a consistent copy of the persistent objects exists at all times at designated storage system servers (called *base nodes*).

MX presents two main technical challenges. One challenge is to balance direct object exchange with transactional consistency in disconnected operation. Another is to support fine-grained updates without introducing false sharing penalties.

False sharing is universally recognized to be a problem in multiprocessor and distributed shared memory systems [7, 19, 20]. But is false sharing a problem for mobile systems? We think that false sharing will become more of a problem for mobile systems for several reasons. One reason is that with mobile computing entering mainstream, users expect their desktop applications to follow them wherever they go. Adapting existing applications to mobile environment is attractive to avoid the cost of re-writing applications from scratch. When modern object-oriented languages, e.g. Java, are used to program persistent object applications, however, it is hard to ensure that non-conflicting accesses do not end up on the same physical page. Moreover, hardware trends indicate that page sizes are likely to grow while object sizes are unlikely to change much. Furthermore, in a disconnected system, the likelihood of conflicting read/write accesses increases as users disconnect for long time periods.

MX has a simple architecture that combines a disconnected client/server system and cooperative caching [13, 18, 22, 45]. Persistent objects are stored in highly-available and reliable repository servers in data centers interconnected by high-bandwidth networks. Less reliable machines, e.g. laptops or PDAs, run disconnected client applications on cached copies of the persistent objects. Like other disconnected storage systems, MX uses optimistic concurrency control [4, 35]. A disconnected client commits tentative transactions on cached data. Upon reconnection, the tentative transactions are validated and, in the case of a conflict, aborted or reconciled.

Disconnected mobile clients form ad-hoc groups that support mobile exchange of data. In a mobile exchange, a client (the helper) provides missing or more current physical pages to a collaborating client (the requester). Since the requester may have read stale objects, the requester validates his tentative transactions against the newer data and merges new and modified pages. The requester manages the bookkeeping needed to commit upon reconnection the modifications based on exchanged data.

MX validation protocols are efficient. The mobile exchange uses a fast coarse-grained validation, falling back to the more expansive fine-grained validation when coarse-grained validation fails. In contrast, the reconnection protocol takes advantage of the invalidations available at the base servers, providing low-cost, fine-grained validation.

To evaluate our techniques, we have implemented an MX prototype and carried out a cost benefit analysis of the mobile exchange performance for a range of shared workloads and a range of ad-hoc network configurations. Preliminary results indicate that the extra cost required to support mobile exchange in a transactional storage system is moderate. Moreover, this extra cost is offset by the cost of accessing remote repository servers over high-latency networks. Nonetheless, some of the mobile exchange benefits can not be quantified. When nearby collaborators can communicate over an ad-hoc network, and have no connectivity to the repository servers, MX enables to accomplish work that would be impossible in other systems.

2 Related Work

Work on cooperative caching systems considers client-to-client access to shared read-only data [13, 18, 22] or shared mutable data [3, 8, 45]. These systems provide direct exchange of data in connected operation.

Recent work in peer-to-peer storage systems, for example [12, 44], considers server side architectures. These systems provide scalable and reliable access to content-addressable shared data and provide limited support for mutable data.

Update-anywhere replication systems support peer-to-peer exchange of shared mutable objects. Many of the approaches utilize the epidemic model [6, 14, 15, 31, 36, 41]. Some of the systems [41] use an optimistic reconciliation-based approach that works in non-transactional single-object domains such as file systems. These approaches only provide weak eventual consistency semantics. Bayou and Lazy Replication [15, 36] take a more pessimistic approach that ensures that all committed updates are serialized in the same order at all replicas using a primary-copy scheme, but they require (for validation) a complete replica of the storage system to be available at each peer, limiting the applicability to small-scale systems. As pointed out by analysis in Gray [24], the traditional update-anywhere approaches suffer from replica divergence as reconciliation fails at individual replicas due to read/write conflicts. Some of the more recent update anywhere replication schemes [6, 27, 32] avoid the replica divergence problem using epidemic quorum techniques that provide strong multi-object consistency and serializability. However, like other epidemic approaches, they require full replication.

Many traditional mobile storages systems in the literature and popular commercial products are based on the client server master copy approach. For example, mobile access to a distributed file system is provided in Coda [33] and Little Work [28] systems, isolation-only transactions for files are considered in [38], The Rover [30] system provides mobile access to general objects, mobile access to persistent objects is described in [9, 10, 25]. A comprehensive survey of existing systems can be found in [29]. These systems provide a consistent master version at the servers but provide no support for client-to-client data exchange.

Oceanstore [34] is a global storage system that deals with access to shared mobile data in wide-area network environment. The architecture is based on nomadic cache pools, similar to MX cache groups, and primary servers. Like MX, Oceanstore uses optimistic concurrency control at the servers to provide strict consistency for persistent data, and a transactional cache consistency protocol [21] in cache pools. Oceanstore does not consider complex data and fine-grained validation.

3 MX

The MX wide-area mobile storage system consists of connected (immobile) base nodes, and mobile collaborating nodes. The base nodes, located in secure data centers connected by high-bandwidth networks, provide reliable and highly-available disk storage for the master versions of persistent objects. A persistent object is owned by a single base node. Since objects may be small (order of 100 bytes for programming language objects [37]), objects on disk are clustered into physical pages. The mobile nodes run client applications accessing the cached copies of the persistent objects. Earlier studies in object storage systems(e.g. [16]) show that for small objects, page-based client caching and page-based transfer from server provide a significant performance advantage compared to single object transfer. Mobile MX client nodes therefore fetch pages from the base nodes and cache pages locally.

The client population is dynamic but not changing fast since access to the shared persistent storage system requires an authenticated account. Mobile client nodes dynamically form collaborating groups, interacting at a close range. We expect a typical group size not to exceed 5-10 clients, smaller groups being the common case. The mobile nodes have the networking capability that allows them to communicate among themselves, and are not limited by the cache capacity.

A typical MX storage system is large: it serves a distributed organization with many branches and mobile employees, such as a large medical research company, or an engineering design firm. Given current technology trends, we expect the system assumptions to be the norm in such organizations.

Before disconnecting from the base nodes, a mobile node pre-loads the cache with up-to-date copies of the master versions of the persistent objects of interest. Techniques similar to hoarding [33] and prefetch queries [25] can be used to assist pre-loading but since the storage system is large, it is not feasible for the mobile nodes to cache the entire storage system.

While disconnected, a mobile node runs tentative transactions accessing the local copies of the cached objects. A tentative transaction records intention to commit, and allows a client to start up a next transaction. Tentative commits lead to "dependent commits" [25, 38, 39]: transaction $T2$ *depends* on $T1$ if it uses objects modified by $T1$ because if $T1$ ultimately aborts so must $T2$. A tentative commit that is not a "dependent commit", defines an "independent action" [23]: a transaction $T2$ that does not use objects modified by $T1$ can commit even if $T1$ ultimately aborts.

At any time the mobile node has access to two cached data versions: a local version and a best-known master version. A tentative transaction updates the local version of the cached data, and generates a standard log record containing modified and new object values to be committed, read and write object sets used for validation, along with undo records that allow to reset the objects to the pre-transaction state. A tentative transaction never modifies the best-known master version.

3.1 Mobile Exchange

A disconnected mobile node can join another mobile peer (or peers) to form a mobile collaborating group. The group *join* protocol identifies the most recent master versions of data available in a group, and notifies the peers with out-of-date versions. A peer (the requester) receiving a notification about an out-of-date versions, reads in the pages from the peer with the most recent master version (the helper) and executes a *catch-up* protocol.

If requesters' transaction has read an outdated master version, it needs to abort. The *catch-up* protocol validates the requesters' tentative transactions against the more recent versions of the data and, if needed, uses the requesters' tentative commit log to undo or reconcile transaction modifications, and install the newer master versions. The requester can then redo the aborted transactions, committing new tentative transactions.

While running within a mobile group, a peer may discover it needs access to an object it does not have. If another peer in the group has a version of this object, the *acquire* protocol provides the missing object directly from that peer. As with catch-up, the acquire protocol provides the best-known master version.

Currently MX only supports the exchange of committed versions. The exchange of uncommitted versions makes the system significantly more complicated. Specifically, it introduces non-local "dependent commits" that reduce the availability of the master versions in mobile environments. Under certain specialized circumstances, exchange of uncommitted changes may be useful, but under most other circumstances the benefits remain unproven. For now, we believe it is more important to support conflict detection and roll-back because these services complement the kind of manual coordination (synchronization) one would expect in collaborative groups.

A mobile node reconnects to the base nodes to commit the tentative transactions. The base nodes validate the transactions using an optimistic concurrency control protocol that checks if a transaction has read the most recent version of the master copies [4]. A tentative transaction that passes validation creates new durable versions of the modified master copies. A transaction that fails validation is aborted at the mobile node that has generated it, and is undone or reconciled. After all the tentative transactions that pass validation commit, the mobile node synchronizes its cache with the base nodes by fetching up-to-date master versions. The tentative transactions that were aborted and undone, can be redone at this point, while the node is connected, or alternatively, redone as tentative transactions after the node disconnects.

3.2 Validation

Keeping track of individual object versions could introduce high system overheads when objects are small. Fetching single objects from a peer during mobile exchange, could increase the cost of communication. Similarly, checking for the most recent master versions of all cached objects in a mobile group, could increase the cost of the mobile exchange. MX addresses this granularity problem by keeping track of page versions. Nevertheless, since the applications modify individual objects rather then entire pages, it is important to avoid the penalty of false sharing. To deal with this problem, MX adds fine-grained validation procedures at mobile exchange and at reconnection.

The mobile exchange protocol identifies page-level conflicts using page versions and then checks objects read from an outdated page by performing a bitwise comparison with the objects in the new page. The approach resembles the adaptive granularity validation schemes [40, 47].

The reconnection protocol takes advantage of the object invalidations available at the servers. The invalidation-based protocol is adapted from [4] and works as follows. Base node directories keep track of which pages are cached at which nodes. When a transaction modifying an object on a page commits, object invalidations are generated for the mobile peers caching that page. When a peer reconnects to validate tentative transactions, the base nodes use the pending object invalidations to check the read sets of the tentative transactions. The invalidations generated for a mobile node are retained at the base node until the mobile node reconnects and completes validation. This assumes mobile nodes reconnect before the resources for retaining invalidations are depleted.

Certain invalidations must be transitive. When a mobile helper node provides a requester with a page the requester does not have, the base node must retain object invalidations for the requester as well. When the helper provides such a page to the requester, it registers this fact, and informs the base node upon reconnection. If the requester node has not yet reconnected, the base node retains the object invalidations until it does.

When peers exchange master versions, persistent objects could get corrupted by a careless peer that scribbles over the cached MX objects. To avoid corrupted objects, master versions can contain embedded checksums created and signed by the base nodes when the new versions of pages are committed. Requesters can verify the checksums for pages they obtain from the helpers in catch-up and acquire requests. Incremental techniques described in [46] can be used to update the checksums inexpensively at commit time, without requiring access to the entire modified page.

Note that because only committed page versions are exchanged, peer group management, i.e. peer node joins and leaves do not interfere with the base transactional system correctness. Appendix A summarizes MX protocol correctness invariants. The invariants guarantee that MX combined with the base invalidation-based protocol [4] provides transaction serializability.

3.3 Undo, Redo and Reconcile

Like other optimistic systems, MX applications need to deal with transactions that abort due to conflicts. The general approach works as follows. When MX detects a serializability conflict, it identifies to the application the actions that violate serializability. Sometimes the application's actions can be undone and redone, and sometimes not. When undo and redo is possible, MX rolls back the object state, invalidates the stale objects, and returns control to the application to redo the transaction. Otherwise, if the action cannot be undone, the application asks the user to compensate in an application-appropriate way.

Applications receive a two levels of support from MX. The MX system itself provides conflict detection in terms of read and write sets, and it provides the ability to restore an earlier pre-transaction state. In addition, MX stores application-specific information provided by the application that identifies the application-level actions associated with each tentative transaction, e.g. the application may identify to MX the portion of the application log recording application operations. The application, however, is responsible for logging the application-level actions associated with the tentative transaction, and for re-executing them if the transaction needs to be redone. The application can also specify to MX when automatic undo and redo is not possible, and interaction with the user is needed. During validation, MX examines each tentative commit record for conflicts. If a tentatively committed transaction has read stale data, MX informs the application, which allows the application to retrieve the application-level actions based on the stale data. Application can choose an appropriate redo action, or a compensating action in place of an undo followed by a redo. For example, the application may consult the user. The comprehensive treatment of application-specific techniques for collaborative redo and reconciliation is outside the scope of this paper. In the rest of the paper we restrict the discussion to applications where undo and redo are appropriate.

Validation and reconciliation can occur under two distinct circumstances: at mobile exchange between disconnected peers, and at reconnection between a mobile host and the base node server. As explained above, disconnected validation produces tentative (non-durable) transactions, which are ultimately validated at reconnection. At reconnection, by contrast, validation produces durable transactions. Similarly, mobile exchange time reconciliation actions are tentative, while reconnection time reconciliation actions are durable.

3.4 Workloads

MX supports peer group access to shared objects. The benefits of MX depend on the patterns of object sharing in the workload. We consider the workloads where we expect MX to be beneficial and workloads where MX offers no benefit. In a disconnected transactional system, obtaining missing shared objects from peers may not be beneficial if peer accesses are conflicting since the tentative transactions accessing the shared object will fail validation and will abort. However, in closely collaborating groups it is more likely that read-write sharing can be manually coordinated by the peers themselves, so that conflicts can be avoided. Nevertheless, for application workloads that exhibit high rates of conflicting accesses, we do not expect MX to be beneficial.

Catch-up allows a peer to obtain more recent objects available at another peer. Catch-up may be expensive if a modifying transaction T at the requester peer has read the outdated page version. In this case validation requires to perform bitwise comparison of objects read from outdated pages, and catch-up potentially requires undo and redo of the modifying tentative transactions. If peer coordination can establish that the requester needs the more recent objects, the undo is not wasted and avoids future aborts.

On the other hand, if peer coordination can establish that the requester transaction T that has read objects from an outdated page has seen up-to-date values (since these objects have not been modified in the more recent page version), the requester can refuse the fine-grained validation and avoid the cost of bitwise comparison. In this case, the fine-grained validation at reconnection time would allow the tentative transaction T to pass validation if T indeed did not read stale objects, and would abort T if it did.

4 Implementation

This section provides some of the MX protocol implementation details. We describe the basic support for disconnected operation, the mobile exchange, and connected and disconnected validation.

MX uses the Thor client/server object-oriented database [37] extended to support disconnected operation as the base storage system. Thor is a good choice because it provides transactional (ACID) storage for objects, supports optimistic concurrency control, supports pages and provides high performance in distributed environments [37].

4.1 Base Storage System

This section outlines the relevant components of the Thor architecture. Thor servers provide persistent storage for objects, clients cache copies of these objects. Applications run at the clients and interact with the system by making calls on methods of cached objects. All method calls occur within atomic transactions. Clients communicate with servers to fetch pages or to commit a transaction.

The servers have a disk for storing persistent objects, a stable transaction log, and volatile memory. The disk is organized as a set of pages which are the units of disk access. The stable log holds commit information and object modifications for committed transactions. The server memory contains cache directory that keeps track of which pages are cached by which clients.

Transactions are serialized using optimistic concurrency control [4]. The client keeps track of objects that are read and modified by its transaction; it sends this information, along with new copies of modified objects, to the servers when it tries to commit the transaction. The servers determine whether the commit is possible, using a two-phase commit protocol if the transaction used objects at multiple servers. If the transaction commits, the new copies of modified objects are appended to the stable log and eventually propagated to the disk.

Since objects are not locked before being used, a transaction commit can cause caches to contain obsolete objects. Servers will abort a transaction that used obsolete objects. However, to reduce the probability of aborts, servers notify connected clients when their objects become obsolete by sending them *invalidation* messages; a server uses its directory and the information about the committing transaction to determine what invalidation messages to send. Invalidation messages are small because they simply identify obsolete objects. Furthermore, they are sent in the background, batched and piggybacked on other messages.

When a client receives an invalidation message, it removes obsolete objects from its cache and aborts the current transaction if it used them. The client continues to retain pages containing invalidated objects; these pages are now *incomplete* with "holes" in place of the invalidated objects. Performing invalidation on an object basis means that false sharing does not cause unnecessary aborts; keeping incomplete pages in the client cache means that false sharing does not lead to unnecessary cache misses. Invalidation messages prevent some aborts, and accelerate those that must happen — thus wasting less work and offloading detection of aborts from servers to clients.

Clients acknowledge all invalidations. The transaction validation protocol treats an acknowledgement as an indication that stale data has been removed from the cache and therefore will not be read by a transaction. Like invalidations, acknowledgements are sent in the background, batched and piggybacked on other messages.

When a transaction aborts, its client restores the cached copies of modified objects to the state they had before the transaction started; this is possible because a client makes a copy of an object the first time it is modified by a transaction.

4.2 Disconnected Operation

To support disconnected operation, we extended Thor to deal with disconnected cache management and tentative commit. The design uses the object management techniques similar to [11] adapted for MX. When a mobile node disconnects it preloads the cache with the most recent copies of the pages potentially needed by the user. Since the storage system can be very large, disconnected cache misses can not be eliminated entirely. Such a miss results in transaction abort and exception propagated to the user.

While disconnected, client applications run tentative transactions that modify the cached copies of pages. When an application issues a commit, the system adds a commit record to the tentative transaction log maintained at the mobile node. The record identifies all the objects read, modified, and created by this transaction, and contains the modified and newly created object values. The commit record also includes the application-level information for dealing with transaction aborts as discussed in section 3.3.

To abort a transaction, the client has to undo the modification made by this transaction, and by all the dependent tentative transactions that read objects modified by this transaction. The client uses the tentative commit log as follows. The value of an object that has been modified by an aborted transaction is restored to the value committed by the most recent preceding tentative transaction in the log. If no such transaction was committed, the value is restored to the value in the most recent master version.

Upon reconnection to the base, MX client sends the tentative commit log records one by one to commit at the server. The client maintains a list of objects undone by aborted transactions so far (undone list). It checks this list against the read and write sets in each subsequent tentative transaction commit record before sending the record for validation to a base node. If an object *read* by the transaction is in the undo list, the transaction is aborted and its modifications are undone; if an object *modified* by the aborted transaction is already in undone list, this value has already been undone and therefore no further undo is needed since the cache contains the correct value.

After the entire tentative transaction log is validated, the client acknowledges all the invalidations sent by the base, re-fetches the invalidated pages. At this point the client interacts with the application in order to redo the aborted transactions using the application level redo information provided by the application.

4.3 Versions

A disconnected mobile peer node can provide pages to other peer nodes. Page version numbers allow the peers to determine which node has the the most recent version of a page. When a version of a page is modified for the first time by a tentative transaction, a clean (unmodified) version stays in an auxiliary cache. Only the clean version of a page are provided to another peer.

The page version numbers are updated by the base node each time a transaction commits modifications to the page. Page version numbers are included with the read sets in the commit records clients send to the base nodes for validation to identify the page version numbers from which the transaction read the objects.

When a mobile node disconnects, the cache directories at the base node keep track of the versions of pages cached in the mobile node at time of disconnection. In addition, object invalidations have the page version number created at the commit time associated with them. If a transaction has read an object from version l and there is pending invalidation for this object associated with page version k, $k > l$, the transaction has read a stale object.

4.4 Catch-up

If a peer discovers that a more recent version of a page it caches is available at a nearby peer, the client requests the page and performs catch-up. If the client has not read the outdated page, the catch-up requires no additional work except reading in the page.

Note however, that since the page versions can increase in the client cache during disconnected operation without the base node knowing this, the client needs to have enough information to determine at reconnection time what were the page version numbers from which a tentative transaction has read. The mapping *TransactionToPage* provides this information. To maintain the mapping, the client first records the initial snapshot of the version numbers corresponding to the cached pages at disconnection time, and then records in the corresponding position in the tentative transaction log the version numbers corresponding to pages acquired from the peers. At reconnect time, the validation procedure uses *TransactionToPage* mapping to reconstruct the correct version numbers for objects read by the transaction and sends them to the base node with the commit record.

At catch-up time, MX validates the tentative transactions that have read or modified outdated pages and the dependent transactions. If a tentative modifying transaction has read an outdated page, it may have read stale objects and modifications may need to abort. On the other hand, if the modified objects read from outdated page were not stale, the newly arrived page needs to be updated to reflect the tentative modifications.

The mapping *PageToTransaction* determines the earliest tentative transaction in the tentative log that has read or modified a given page. To perform fine-grained catch-up validation, the client scans the log from the earliest transaction that read an outdated page. Client performs a bitwise comparison of an object that was read from the page (using the master version of the outdated page in case the object has been modified by later transactions) comparing it to the object in the new page version. If the object have changed, transaction gets aborted together with the dependent transactions. If the bits have not changed, the newer version of the page and the modifications are merged.

In the case when the bitwise comparison provides a false negative, i.e. no change is detected in the bits because the object has been modified and later reset to the same value, the connected validation procedure using invalidations will detect the conflict since the mapping *TransactionToPage* reconstructs at commit time the actual version number of the page from which the transaction read.

After undoing the aborted tentative transactions, the system interacts with the application to redo the aborted transactions using the information provided by the application as explained in Section 3.3. The new portion of the log, generated for the redone transactions, includes the appropriate new page version numbers records.

Compared to the invalidation-based reconnection-time validation at the base server, the disconnected validation introduces the cost of bitwise object comparison and page merging. The cost depends on the workload, specifically, the number of the different objects read and modified on outdated pages, and the sizes of the objects.

4.5 Acquire

A cache miss at one peer in a group can be served by acquiring the page from another peer cache. Since the base node cache directory is unaware of the acquired page, to support fine-grained validation for the transactions that access such an acquired page, the client maintains information on sending and acquiring missing pages and reports this information to the base nodes at the reconnect time. This information is used to efficiently generate invalidations for the acquired missing pages and to garbage collect the invalidations when they are no longer needed.

4.6 Object Creation and Deletion

Object creation and deletion can present a problem for disconnected protocols [36]. We briefly consider how creation and deletion interacts with the mobile exchange. MX follows the proposal in [11] for disconnected object creation and deletion. New objects created by a disconnected MX client only become persistent when committed at the base servers. Since in MX uncommitted data is never provided in a mobile exchange, object creation poses no additional issues for MX.

Objects are never explicitly deleted in MX because, as in Thor, MX uses the persistence by reachability model and relies on a transactional garbage collector to reclaim the free storage. An object, unreachable in the persistent object graph, may become tentatively persistent if a disconnected client commits a tentative transaction that creates a reference to it. Therefore, the garbage collector considers all objects cached in disconnected clients to be the roots of garbage collection. To support mobile exchange, the garbage collector needs to track where objects are cached. It uses the page tracking technique similar to the one used by the validation protocol to generate invalidations. The disconnected garbage collector has not been implemented yet.

4.7 Hooking up with a Peer

To hook-up with a collaborating peer, a node starts up a connector module that runs a simple discovery procedure that lets the client to join an existing group, or start a new group. The discovery procedure makes use of the INS intentional naming system [2]. The details of the discovery protocol, and of the group management protocol dealing with client joins and leaves, are omitted for lack of space.

The connector module, available in each peer, coordinates the page transfers for catch-up and acquire. At the hook-up time a peer sends its page table with the version numbers to the connector who compares the version numbers and sends out to each peer the list of pages available for catch-up. A peer with an outdated version reads in the more recent versions by fetching them via the connector module. Hook-up cost increases with the group size, but for the small groups typical in MX, we do not expect the cost to be significant.

To support page acquire, the connector maintains a directory of pages available in the peer group. While a peer runs connected to a group a cache miss experienced by a peer reaches the connector, and if the missing page is available in the directory, the request is forwarded to the helping peer. This scheme is very similar to one used in the BuddyCache [8] system.

5 Performance Evaluation

We have implemented an MX prototype and conducted a performance study to evaluate the costs and benefits of mobile exchange, the novel feature in MX. Our experiments highlight a collaborative situation when a mobile team operates far away from base servers because this is the typical environment MX is designed for. Nevertheless, an MX client sometimes needs to run in a non-collaborative situation. To support mobile exchange, MX introduces extra mechanism at the mobile peer nodes and the base nodes. We evaluate the cost of the extra mechanism a client needs to pay in MX for being mobile exchange enabled.

In a collaborative situation mobile exchange can provide a benefit that can not be measured: the ability to obtain data and continue work while disconnected. However, mobile exchange can also provide a performance benefit that can be measured: fetching missing or more recent data from a nearby peer, instead of reconnecting to the remote repository. The benefit is similar to cooperative caching but, because of the validation, the overheads are different. We evaluate this benefit.

The evaluation compares the performance of MX to the performance of a mobile transactional object storage system without the mobile exchange (Mobile). To evaluate the overheads, we analyze the costs of MX protocol components and consider the effect of varying sharing in the workload. To evaluate the benefits, we compare the time it takes to complete a task using MX and Mobile in different peer network confgurations.

The results indicate that in a transactional object storage system the extra cost required to support mobile exchange and fine-grain validation is moderate. Furthermore, the extra cost is offset by the cost of accessing the base repository when network latency is high.

5.1 Experimental Setup

The experiments use two system configurations: the *MX* system runs our implementation of MX protocols in Thor system, extended to support disconnected operation as described in Section 4.2, the *Mobile* system runs Thor extended to support disconnected operation but without the support for mobile exchange.

Unfortunately, there is no standard benchmark available for collaborative systems at this time. We are considering to develop such a benchmark in our future work. Our workloads are based on the multi-user OO7 benchmark [17]; this benchmark is intended to capture the characteristics of complex data in many different CAD/CAM/CASE applications, but does not model any specific application. We use OO7 because it allows to control the sharing of complex data and because it is a standard benchmark for measuring object storage system performance. Appendix B contains a detailed decription of our implementation of the multi-user OO7 benchmark with the traversals T1 (read-only), T2a (read-write sparse modifications), and T2b (dense modifications).

Our experiments compare the performance costs of MX and Mobile when running tentative transactions, and validation, fetch and commit operations. These costs are determined by either the actions of a single peer or the interactions between two peers, the requester and the helper in MX, and a peer and a base node in Mobile. The costs are independent of the size of the collaborative group for the small group sizes expected in MX. We use therefore a system configurations containing a base node and two peers in our experiments.

The server (base node), and the two clients ran each on a 600MHz Intel Pentium III processor based PC, 128MB of memory, and Linux Red Hat 6.2. Unless otherwise specified, the clients were connected by a 100Mb/s Ethernet; the long distance connection to the base node used an estimated network latency of 70 msec (70 milliseconds) and 1 Mb/s connection approximating a cable modem. The OO7 database was stored on the server disk; in all the experiments the fetched data was cached in the server cache. The experiments ran in Utah experimental testbed emulab.net [1] on a dedicated, unloaded and isolated system.

5.2 Clean Versions

Compared to the Mobile system, MX requires extra space and CPU for the clean versions of modified pages, and extra space in the tentative transaction log to store the peer exchange records identifying page versions for the $TransactionToPage$ mapping. The clean versions are by far the main space expense. The CPU cost of creating a clean copy of a page including the cost of page copy, and unswizzle references, averages 0.32 msec in our workloads.

5.3 Acquire

To evaluate the costs and benefits of acquiring missing objects using MX in wide-area environment, consider the scenario described in Section 1 where a team of design engineers (here called Mary and John), travels to present a design proposal to another company (called FarAway), and needs to revise the joint design. In MX, after coordinating the revision details, John acquires Mary's proposal data directly from Mary, and produces a draft modifying the documents to produce an internally consistent proposal. After the sales presentation, the proposal gets revised some more, until John returns home, reconnects to the repository and commits the revised version. In $Mobile$, the team actions are identical, except to acquire the proposal data, John fetches it remotely from the home repository. We compare the time to accomplish the task in MX and Mobile.

To model the above scenario in MX, Mary runs a cold T1 traversal (read-only) pre-loading her laptop before disconnecting. John disconnects without the objects. At the $FarAway$ company site, it takes Mary and John time $HookUp$ of 2 msec to hook up (i.e. send page tables to the connector and receive (empty) stale page lists), and it takes John time $MXCold$ of 2.58 sec to run a cold T1 $(50, 0, 50, 0)$ traversal fetching from Mary missing objects contained in 1255 pages. To commit 100 tentative transactions running a hot T2a $(0, 50, 0, 50)$ traversal (sparse modifications), John needs time $HotTentative$ of 35.01sec. T2a $(0, 50, 0, 50)$ single traversal takes 0.234 sec, single tentative commit 0.0989 sec. For comparison, T2b $(0, 50, 0, 50)$ traversal (dense modifications) takes 0.243 sec, tentative commit time 0.147 sec (not shown). Tentative commit is expensive in our (unoptimized) implementation of the disconnected system because it includes copying read and write sets, and object update and undo information into the log. We are working on optimizing these costs. To commit the modifications at home repository on a fast local network John executes the invalidation generation step that takes time $InvalGen$ of 1.31 msec, and validates and commits in time $Commit$ of 2.29 sec.

In the $Mobile$ system, John reads in over the long distance network the project data in time $MobileCold$ of 296.911 sec, runs 100 tentative T2a $(0, 50, 0, 50)$ transactions in time $HotTentative$ of 33.29 sec, and validates and commits in time $Commit$ of 2.29 sec. Running tentative transactions including modifications is more expensive in MX than in Mobile because of the cost of creating a clean version of a modified page. The reason it takes John such a long time to get Mary's objects in Mobile is that his application is acquiring the objects by faulting the missing pages synchronously. If the missing pages are known in advance, John can prefetch Mary's objects asynchronously in a pipeline constrained only by the bandwidth of the link to the base repository. Section 5.4 discusses pipelined fetching. Nevertheless, in general applications, dynamic prefetch is known to be a hard problem.

Table 1 summarizes the results. The results indicate that the overhead of acquiring missing objects including extra validation costs, slower tentative transactions and the cost of hookup, is offset by the cost of acquiring the missing objects from the home repository when the latency of access to the repository is high. Of course, in situations when the access to remote repository is impossible, this may be the only available choice.

5.4 Catch-up

To evaluate the performance of catch-up, we consider a variation of the scenario where Mary and John are making collaborative changes at the $FarAway$ company site. Mary loads shared proposal objects into her laptop, disconnects, and travels to the $FarAway$ company. John stays behind, commits last minute modifications to the proposal in response to some late breaking news, and disconnects to travel to $FarAway$. At $FarAway$, when John informs Mary about his changes, he finds out Mary has already made some tentative modifications to the proposal. Mary suggests she gets John's modifications and they coordinate the next set of changes to prevent future conflicts.

In MX, Mary hooks up with John, performs catch-up with undo and redo since her changes were based on stale objects. In $Mobile$, to get $John's$ modifications, $Mary$ needs to reconnect to the base, abort the tentative transactions that accessed objects modified by John, refetch the modified pages, and redo her tentative modifications.

To model this scenario, in the MX experiment $Mary$ runs cold T1 $(50, 0, 50, 0)$ traversal (read only) to pre-load cache before disconnecting. Then $John$ runs connected T2a $(0, 0, 0, 100)$ transactions modifying the shared module and commits. At the $FarAway$ site $Mary$ commits 50 tentative transactions T2a $(40, 40, 10, 10)$ while running disconnected in time $HotTentative1$ of 15.64 sec. It takes Mary and John 11.49 msec to hook up and it takes Mary the time of $FetchMx$ of 0.76 sec to read in from $John$ the 532 modified pages. It takes Mary time $ValidateMX$ of 4.61 sec to catch up. 24% of Mary's transactions get aborted in this workload. To redo her changes, Mary commits 12 tentative transactions T2a $(40, 40, 10, 10)$ while running disconnected in time $HotTentative2$ of 3.75 sec.

In $Mobile$, to obtain John's modifications, Mary needs to reconnect to the remote home repository. She receives invalidations, validates and aborts the tentative transactions that accessed stale pages modified by John(24% aborts) in time $ValidateMobile$ of 4.00 sec (local validation time). Mary fetches the 532 modified pages in time $FetchMobile$ of, 115.146 sec and re-executes and commits 12 tentative transactions T2a $(40, 40, 10, 10)$ while running disconnected in time $HotTentative2$ of 3.75 sec.

Here, unlike in the scenario in Section 5.3, Mary could easily pipeline fetches of the invalidated pages, since the list of invalidated pages is known after the validation and abort. We do not measure pipelined fetches because our system currently supports only a limited form of prefetch in the form of clustered pages that takes advantage of object clustering for both fetching from a peer ($FetchMX$) and from base ($FetchMobile$). The pipelined fetch costs for Mobile, computed assuming a 1 Mb/s cable modem are included for comparison. Table 2 summarizes the results.

Table 1. Costs of Acquire

Time (sec)	MX	MOBILE
MobileCold	0	296.911
HookUp	0.002	0
MXCold	2.58	0
HotTentative	35.01	33.29
InvalGen	0.00131	0
Commit	2.29	2.29

Table 2. Costs of Catch-Up

Time (sec)	MX	MOBILE
HotTentative1	15.64	15.64
ValidateMobile	0	4.00
HookUp	0.012	0
ValidateMX	4.61	0
FetchMX	0.76	0
FetchMobile	0	115.146(33.39 piped)
HotTentative2	3.75	3.75
Total	24.77	138.8(57.04 piped)

Validation Costs. Table 3 breaks down the MX catch-up costs ($ValidateMX$ in table 2). These are read set check ($WithBitwiseCompare$), including checks against the set of undone objects ($2,283,816$ objects checked) and bitwise comparison for objects read from the stale pages (0.257 sec for $276,385$ objects checked); undo of aborted transactions using the log ($Undo$); tracking committed modifications on new pages (CollectUpdates), and merging the modifications with the new pages ($Merge$). To avoid repeated swizzling, unmodified objects (that did not change size) are currently merged from the new page into the old swizzled page. Read set check dominates the catch-up cost; the check against the undone objects beeing the main cost in this workload.

Table 4 breaks down the validation with base costs ($ValidateMobile$ in Table 2) that include read set check $WithInvalidations$ against the invalidations of the cached shared pages, and against the set of undone objects; and undo of the aborted transactions using the log ($Undo$). The $Undo$ costs in Mobile and MX are different, because MX uses a somewhat different undo protocol that takes

Table 3. Validation with a Peer

Validation Action	Time(sec)
WithBitwiseCompare	4.41
CollectUpates	0.014
Merge	0.13
Undo	0.0044
Other	0.052
Total	4.61

Table 4. Validation with Base

Validation Action	Time(sec)
WithInvalidations	3.98
Undo	0.0061
Other	0.014
Total	4.00

advantage of merging. As in MX, the read set check dominates the validation cost.

Sharing. When transactions access many objects, failed validation (with undo and merge) may take less time than successful validation since detection of a conflict terminates the remaining check, while successful validation completes the entire read set check. Figure 1 shows this trend as conflict rate increases for workloads with sparse (T2a) and dense (T2b) modifications for validation with base ($Mobile$) and peer (MX). Validation costs are higher for T2b because of larger undo sets and increased undo cost. Validation cost increases initially, because for low abort rates the size of the undo set dominates the cost of the validation. As abort rate increases, the benefit of avoided checks due to aborts dominates the total validation cost.

In above experiments the difference between the validation cost with a peer and with a base is small. In general, the difference between validation with peer and with base may depend on the page clustering and the amount of false sharing in the workload, parameters that are not directly controlled in the standart 007 benchmark.

To evaluate the effect of false sharing, we created a modifed 007 database and two new traversals derived from T2b. In the modified datatabase, the atomic parts and conections in a composite part are numbered and connected in a way that allows the new traversal $T2b_{odd}$ ($T2b_{even}$) to only read and modify the odd (even) atomic parts. In the scenario with John and Mary in Table 2, when John commits modifications to the database using $T2b_{odd}$ traversal, and disconnected Mary concurrently runs tentative $T2b_{even}$ traversals, the transactions do not conflict but the workload contains false sharing since the odd and even parts share the same pages. Figure 2 compares the time it takes Mary to validate with John in MX, and with base in Mobile, after Mary commits 50 tentative $T2b_{even}$ transactions. The experiment measures validation costs for different amount of false sharing by controlling in the validation procedure for a fixed size transaction read set, the percentage of objects Mary reads and writes on pages modified by John. Note, that validation costs here differ from ones in Table 2 because a tentative T2a transaction reads more and modifies less objects then the tentative $T2b_{even}$ ($T2b_{odd}$) transactions ($T2b_{odd}$ reads $1,406,203$ objects and modifies $61,470$ objects on 1199 pages). The cost of validation increases for both

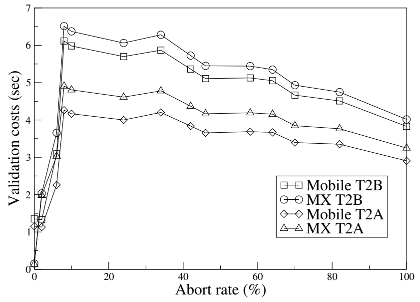

Fig. 1. Workloads with aborts

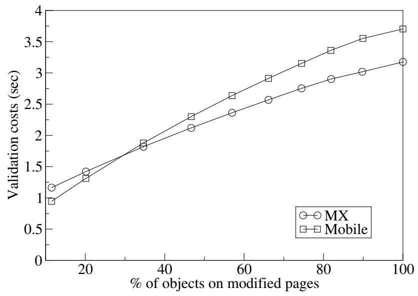

Fig. 2. Workloads with false sharing

Mobile and MX as more objects are read and modified on pages containing false sharing since more objects need invalidation-based checks in Mobile and bitwise comparison checks in MX. The cost of the check against the undo set and the coarse-grain check against modified pages performed for the entire read-set is the same in Mobile and MX and is fixed in this experiment since the read-set size is fixed. The invalidation check cost is higher then bitwise comparison and update merge cost when many objects are checked and invalidation sets are large since merge costs do not increase as more objects are modified in our implementation. Overall, the experiment indicates that for OO7 workloads with and without false sharing, the cost of validation with the peer is comparable to the invalidation-based validation with the base.

5.5 Performance Benefit

The performance benefit of fetching from peer instead of base depends on peer connectivity. The experiments so far assumed co-located peers connected by 100 Mb/s Ethernet. Below we consider this benefit in other network configurations. The benefit is defined as the time difference between fetching from the base (in $Mobile$), and from the peer (MX) using acquire or catch-up, relative to the fetch from the base. The MX time when using catch-up ($Catch - up$) includes hook-up, fetch from peer, and validation; the MX time using acquire ($Acquire$) include no validation. The scenario corresponds to Table 2 with 532 modified pages fetched and validation costs $ValidateMX$.

Figure 3 shows the benefit for co-located peers connected by a wireless 801.11 ad-hoc network with a 5 millisecond latency and a range of bandwidth corresponding to varying distance between peers. Both the measured benefit using non-piped fetches, and the computed benefit assuming piped fetches for MX and Mobile are shown. The results indicate that with non-piped fetches e.g. when prefetch information is unavailable for acquire, fetch from the peer has a high 60%-90% performance benefit. With piped fetches, the catch-up validation costs limit the benefit when the peer network bandwidth is below 2 Mb/sec. Above 2 Mb/sec fetching from peer provides a substantial $40\% - 80\%$ benefit.

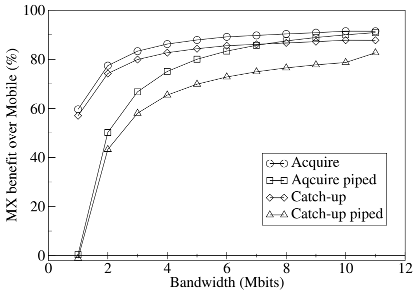

Fig. 3. MX benefit: wireless ad-hoc network

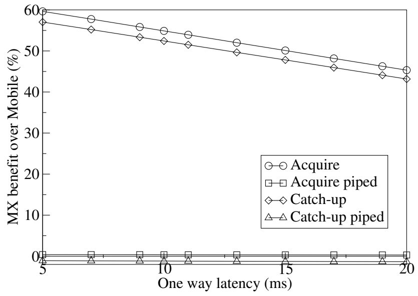

Fig. 4. MX benefit: wireless via Internet

To show the limits of the benefit, the next experiment considers MX in a configuration it was not originally intended for, where instead of beeing co-located, mobile peers roam within a metropolitan area using a 1 Mb/sec wireless 801.11 network connection to base stations interconnected via a metropolitan Internet. The results in Figure 4 show that with typical metropolitan Internet latencies, fetch from the peer provides no performance benefit even for acquire (no validation) unless non-piped fetches are used because prefetch information is unavailable.

In summary, the performance benefit results indicate that unless peer network has good connectivity, fetch and validation with a peer is rather costly. Validation with a home may be preferable because it is less expensive and because it insures transactions indeed commit. However, fetch and validation with a peer is attractive on a remote location provided peer coordination insures the validation is not wasted due to conflicting accesses.

6 Conclusion

Direct exchange of shared objects is useful for mobile collaborators. Mobile storage systems address the issue of how to do this for simple data but efficient direct exchange of complex data remains an open problem. This issue will become even more important as mobile computing becomes pervasive, and mobile use of collaborative applications manipulating complex data becomes common.

The paper describes MX, a new mobile caching system for collaborative applications, that provides the first systematic approach to solving this problem. MX clients can obtain missing or more recent objects from nearby collaborators. MX validates the exchange, manages the bookkeeping to allow modification based on exchanged data to commit upon reconnection, and insures that a consistent master copy of the persistent objects exists at all times at designated repository servers. MX supports well the exchange of complex data because it provides efficient (page-based) data transfer and fine-grained object validation that avoids the problem of false sharing. Moreover, the validation protocols do not restrict repository size because MX is a caching system and does not require full replication. The paper describes MX caching and validation protocols and specifies the invariants that insure correctness of the mobile exchange.

The paper also describes a prototype implementation based on the Thor object storage system. Preliminary performance results using the OO7 benchmark indicate that the extra validation cost required to support mobile exchange is moderate. For nearby collaborators communicating over a typical ad-hoc network this extra cost is offset by the cost of accessing repository servers over high-latency networks, making the exchange attractive on remote locations. The MX scheme does have some costs. The cost of catch-up with the more recent shared objects is proportional to the number of objects read from the outdated pages. Nevertheless, when members of a collaborating team communicate over ad-hoc network and have no global connectivity, obtaining shared objects from a collaborator may allow to accomplish work that would be impossible in other systems.

Our paper makes the following contributions:

- It describes a new technique, mobile exchange, that allows mobile collaborating clients to acquire from each other complex data residing in large-scale repositories.
- It presents fine-grained transaction validation protocols for the exchanged objects.
- It describes a prototype implementation that supports mobile exchange in a transactional object storage system.
- It presents a performance study of the costs and benefits of mobile exchange for a range of shared data workloads and a range of client network configurations.

7 Acknowledgements

We thank Jay Lepreau and the staff of Utah experimental testbed emulab.net [1], especially Leigh Stoller, for their help with the testbed, Sidney Chang and Dorothy Curtis for the help with implementing object referencing in disconnected operation, and Jim Gray and Maurice Herlihy for helpful comments.

References

1. 'emulab.net', the Utah Network Emulation Facility. supported by NSF grant ANI-00-82493.
2. W. Adjie-Winoto, E.Schwartz, H.Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM SOSP*, 1999.
3. A. Adya, M. Castro, B. Liskov, U. Maheshwari, and L. Shrira. Fragment Reconstruction: Providing Global Cache Coherence in a Transactional Storage System. In *Proceedings of the International Conference on Distributed Computing Systems*, May 1997.

4. A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1995.
5. A. Adya and B. Liskov. Lazy Consistency Using Loosely Synchronized Clocks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, August 1997.
6. D. Agrawal, A. E. Abbadi, and R. Steinke. Epidemic Algorithms in Replicated Databases. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, May 1997.
7. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. In *IEEE Computer, Vol. 29, No. 2*, February 1996.
8. M. Bjornsson and L. Shrira. BuddyCache: High-Performance Object Storage for Collaborative Strong-Consistency Applications. In *OOPSLA Conference*, November 2002.
9. M. Butrico, H. Chang, A. Cocchi, N. Cohen, D. Shea, and S. Smith. Gold Rush: mobile transaction middleware with Java-object replication. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, June 1997.
10. S. Chang and D. Curtis. An Approach to Disconnected Operation in an Object-Oriented Database. In *Proceedings of Third International Conference on Mobile Data Management, Singapore*, January 2002.
11. Sidney Chang. Adapting Object-oriented Database for Disconnected Operation, M.Eng. Thesis. Technical report, Massachussetts Institute of Technology, May 2001.
12. Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *ACM SOSP*, October 2001.
13. M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using Remote Client Memory to Improve File System Performance. *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.
14. A. Demers, D. Gren, C. Hauser, W. Irsh, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC)*, August 1987.
15. 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. *Proceedings of the Workshop on Mobile Computing Systems and Applications*, December 1994.
16. D. DeWitt, P. Futtersack, D. Majer, and F. Velez. A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems. In *The Sixteenth International Conference on VLDB*, August 1990.
17. M. Carey et al. A Status Report on the OO7 OODBMS Benchmarking Effort. *Proceedings of OOPSLA Conference*, October 1994.
18. M. Feeley, W. Morgan, F. Pighin, A. Karlin, and H. Levy. Implementing Global Memory Management in a Workstation Cluster. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
19. M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Integrating coherency and recoverablity in distributed systems. In *Proceedings of the First Usenix Symposium on Operating sustems Design and Implementation*, May 1994.

20. P. Ferreira and M. Shapiro et al. PerDiS: design, implementation, and use of a PERsistent DIstributed Store. In *Recent Advances in Distributed Systems, LNCS 1752, Springer-Verlag*, October 1999.
21. M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems*, September 1997.
22. Michael Franklin, Michael Carey, and Miron Livny. Global Memory Management for Client-Server DBMS Architectures. In *Proceedings of the 19th Intl. Conference on Very Large Data Bases (VLDB)*, August 1992.
23. D. Gifford and J. Donahue. Coordinating Independent Atomic Actions. In *Proceedings of IEEE COMPCON Digest of Papers*, February 1985.
24. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of ACM SIGMOD Conference*, June 1996.
25. R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected Operation in the Thor Object-Oriented Database System. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.
26. M. Herlihy and M. McKendry. Timestamp-Based Orphan Elimination. *IEEE Transactions on Software Engineering*, July 1989.
27. J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi. Epidemic Quorums for Managing Replicated Data. In *Proceedings of the 19th IEEE International Conference on Performance, Computing, and Communication*, February 2000.
28. L. B. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location Independent Computing*, August 1995.
29. J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2), Jun 1999.
30. A. Joseph, A. Lespinasse, J. Tauber, D. Gifford, and F. Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
31. L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *Proceedings of the ACM CSCW Conference*, September 1988.
32. P. J. Keleher. Decentralized Replicated-Object Protocols. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, May 1999.
33. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM TOCS*, February 1992.
34. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
35. H. Kung and J. Robinson. On optimistic methods for concurrency control. In *ACM TODS 6(2)*, June 1981.
36. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. In *ACM TOCS 22(3)*, November 1992.
37. B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing Persistent Objects in Distributed Systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, June 1999.
38. Qi Lu and M. Satyanarayanan. Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions. In *Fifth IEEE HotOS Topics Workshop*, May 1995.

39. W. Montgomery. Robust concurrency control for distributed information system. Technical report, MIT Laboratory for Computer Science,Cambridge, December 1978.
40. L. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence For Intermittent Connectivity. In *Usenix Summer Technical Conference*, June 1998.
41. T. W. Pager, Jr. R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, G. H. Kuenning A. Goel, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software - Practice and Experience*, February 1998.
42. K. Petersen, M. J. Spreitzer, M. M. Theimer D. B. Terry, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operationg Systems Principles*, October 1997.
43. S. Phatak and B. Badrinath. Conflict Resolution and Reconciliation in Disconnected Databases. In *MDDS*, September 1999.
44. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *ACM SOSP*, October 2001.
45. P. Sarkar and J. Hartman. Efficient Cooperative Caching Using Hints. In *Proceedings of the Usenix Symposium on Operation Systems Design and Implementation*, October 1996.
46. Liuba Shrira and Ben Yoder. Trust but Check: Mutable Objects in Untrusted Cooperative Caches. In *POS8*, Sept 1998.
47. M. Zaharioudakis, M. J. Carey, and M. J. Franklin. Adaptive, Fine-Grained Sharing in a Client-Server OODBMS: A Callback-Based Approach. *ACM Transactions on Database Systems*, 22(4):570–627, December 1997.

8 Appendix A: MX Invariants

Below we summarize MX protocol by stating its invariants. The MX protocol maintains the following simple invariants I1 and I2 that insure protocol correctness. In the following, $Max(C, i)$ ($Min(C, i)$) denote the highest (lowest) master versions of the page C possibly cached at node i since disconnection, $Master(C)$ is the master version of C at the base node, and $Invalid(C, i)$ is the set of invalidations of C retained at the base node since node i disconnection. We say a tentative transaction $T2$ *depends* on a tentative transaction $T1$ if $T2$ reads objects modified by transaction $T1$.

Invariant I1. If a mobile node i has provided a page C to a mobile node j, then either node j has disconnected without the page C, or $Max(C, i) > Max(C, j)$ at the time of the exchange.

Invariant I2. Let i be a mobile node reconnecting to a base node with a page C, where $Min(C, i) = k$, and i has either disconnected with C, or has disconnected without it and has acquired C from a mobile node j. If at reconnection time at the base node $Master(C) = m$, $k < m$, then $Invalid(C, i)$ contains all the invalidations for C that correspond to versions $k + 1$ through m of C.

Invariants $I1$ and $I2$ insure that if a tentative transaction T at the mobile node i has read an object x from a master version l of a page C while disconnected, and i reconnects with $Min(C, i) <= l <= Max(C, i)$ and T passes validation, then value x read by T is not stale. Moreover, if at validation time none of the objects read by T are stale and T does not depend on a tentative transaction S that has aborted, T will pass validation.

MX protocol insures that the master versions at the base nodes reflect a state generated by a globally serializable sequence of update transactions. Tentative transactions that fail validation and need to abort, however, may observe an inconsistent state if they access objects on multiple pages. This is undesirable for applications that rely on the invariants maintained by the transactions. This problem, related to the orphan detection problem in distributed systems using locking [26], and lazy consistency problem in optimistic systems [5] has efficient solutions in connected environments using timestamps and safe reconnection intervals that are not applicable for the disconnected environment.

One possible solution is to require mobile hosts to hoard page sets that are "invariant-complete" i.e. contain all objects in a multi-object invariant. At disconnect time, the hoarding procedure runs as a read-only transaction to make sure the hoarded set reflects a transactionally consistent "invariant-complete" state. Since mobile exchange always involves "invariant-complete" set of pages, tentative transactions will always observe a transactionally consistent state.

9 Appendix B: The OO7 Benchmark

The OO7 database contains a tree of assembly objects with leaves pointing to three *composite* parts chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic* parts, accessible from a single root atomic part, and linked by *connection* objects; each atomic part has 3 outgoing connections. We use a *small* database that has 20 atomic parts per composite part. The multi-user database allocates for each client a "private" module consisting of one tree of assembly objects, and adds an extra "shared" module that scales proportionally to the number of clients. To run experiments with a 2 client team we generate OO7 database with modules for 2 clients. The objects in the database are clustered in 8K pages, which are also the unit of transfer in the fetch requests. We expect a typical MX configuration to not be cache limited and therefore focus on workloads where the objects in the client working set fit in the cache.

The OO7 workload consists of two types of transactions, *read-only* and *read-write*. Read-only transactions use the T1 traversal that performs a depth-first traversal of entire composite part graph. Read-Write transactions use the T2b traversal that is identical to T1 except it modifies all the atomic parts, or the T2a traversal that modifies only the root part. A T1 traversal accesses 52389 objects if it traverses a private module, and 55924 objects if traverses the shared module. T2a modifies 1064 objects (on 532 pages) and T2b modifies 21280 objects (on 613 pages) when traversing the shared module. A transaction includes one traversal.

A OO7 client workload is specified by a $4 - tuple$ that defines the mix of the read-only and read-write transactions accessing the private and the shared modules of the database. E.g. a $(40, 40, 10, 10)$ $4 - tuple$ defines a workload with 40% read-only, 40% read-write transactions in the private module and 10% read-only, 10% read-write transactions in the shared part. Note that read-write transactions do no modify all the objects it accesses e.g. T2b modified 49% of the objects it accesses. The level of contention (conflict) in the workload is controlled by the read-write accesses to shared module.