Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers*

Linha Shrir<sup>1</sup>, Hong Tian<sup>2</sup>, and Doug Terry<sup>3</sup>

<sup>1</sup> Brandeis University
<sup>2</sup> Amazon.com
<sup>3</sup> Microsoft Research

Abstract. Exo-leasing provides a new way to implement escrow synchronization for disconnected applications. A key problem facing disconnected applications is the need to coordinate concurrent operations on shared data objects to avoid conflicts. Escrow synchronization is a well-known technique, useful for inventory control, that avoids conflicts by taking into account the semantics of fragmentable object types. Unfortunately, current techniques cannot be used on generic “commodity” servers because they require the servers to run the type-specific escrow synchronization code. This is a severe limitation for systems that require application-specific synchronization but rely on generic components to exploit economies of scale.

Our exo-leasing method provides escrow synchronization without running any type-specific code in the servers. Instead, escrow synchronization code runs in the client, resulting in a modular system with the ability to use generic commodity servers. Running synchronization code in the client provides an additional benefit. Unlike any other system, our system allows a disconnected client to avoid conflicts by coordinating with another disconnecting client, reducing the need to coordinate with the servers. Measurements of a prototype system indicate that our approach achieves escrow-based conflict avoidance at moderate performance overhead on common expected workloads.

1 Introduction

Mobile collaborators wish to continue their collaborative work wherever they go. In spite of improving network connectivity, wide-area connectivity cannot be taken for granted because of physical, economic and energy factors. Moreover, the increasing trend towards storing data in utility data centers is making it harder for mobile workers to share and access their data while out of the office. It is useful, therefore, to develop techniques that enable mobile users to continue collaborative work while disconnected and operate independently without compromising data consistency.

Disconnected access to shared data is by now commonly supported via a well understood process [18, 17]. A mobile client pre-loads objects before disconnecting and optimistically manipulates locally-cached copies of objects, periodically reconnecting to validate the changes against a “master copy” of data stored reliably at the storage server. If the validation detects conflicts, the client has to abort the changes or reconcile them, possibly using application-specific resolvers [21, 32].

The penalty for aborts and after-the-fact conflict resolution, however, may be too high in some applications. For example, a mobile salesman may accept customer orders based on cached, but

* This research was partially supported by NSF grant CNS-0427408. The work was done while Hong Tian was at Brandeis University and Liuba Shrir<sup>1</sup> was at Microsoft Research, Cambridge, UK.
out-of-date information only to discover, upon returning to the office, that the purchased items are out of stock, thereby resulting in cancelled orders and unhappy customers. To avoid costly conflicts, a mobile client, before disconnecting, can obtain reservations \cite{30} that guarantee (in-advance) the successful completion of specific transactions while disconnected. Escrow synchronization \cite{26} is a well-known scheme useful for inventory control, that provides such reservations exploiting the properties of fragmentable \cite{40} object types. It allows disconnected mobile clients to avoid conflicts while independently making changes to shared objects \cite{30}. For example, members of a mobile sales team can each obtain a reservation for a portion of the available sales items and independently validate sales transactions while disconnected.

Current escrow synchronization techniques suffer from a limitation that precludes the use of generic "commodity" servers because they require type-specific escrow synchronization code at the servers. Most data centers will not allow customers to run unproven custom code on shared storage system servers for performance and security reasons. This is a problem for systems that require application-specific synchronization but rely on generic components to exploit economies of scale (for example, "cloud computing" systems are likely to have this problem).

The contribution of our work is to fix this limitation. This paper describes a new technique called exo-leasing that implements escrow synchronization without running type-specific code at the servers. Instead, the type-specific code that implements escrow synchronization runs in the client. This way, new applications using escrow and other fragmentable object types can be developed without modifying the servers. The result is a modular system with the ability to use commodity servers.

An additional benefit of running the synchronization code at the client is the ability to provide new functionality. Unlike any other system, our system allows to "split" escrow reservations among disconnected clients. A disconnected mobile salesman on a sales trip can transfer some of his reservations to a partner. Exo-leasing makes the transfer possible because the synchronization code running at the client encapsulates the complete synchronization logic. Disconnected reservation transfer reduces the need to communicate with the servers, providing a complementary benefit to disconnected cooperative caching \cite{35,5,28} that transfers data but not reservations.

We implemented MobileBuddy, an escrow synchronization system built on top of a generic transaction system using exo-leasing, and evaluated the performance overheads introduced by our techniques. Measurements indicate that if the client obtains reservations but does not benefit from them, our techniques impose a moderate performance penalty. If the client benefits from conflict avoidance, enabled by the reservation and reservation transfer, the cost is reasonable since conflict avoidance saves work.

To summarize, this paper attacks an important insufficiently studied problem in mobile computing space: namely, how a commodity storage system can support escrow synchronization so that disconnected client applications can control shared inventory data while avoiding conflicting updates that later need to be aborted or resolved. The paper makes the following contributions: 1) It introduces exo-leasing, a new modular approach that combines escrow reservations with optimistic concurrency control. By offloading the type-specific escrow code from the servers to the clients, it provides the ability to use commodity servers, making escrow reservations practical in commodity storage systems. 2) It describes an escrow reservation transfer facility enabled by exo-leasing, including the definition of transfer semantics, and new transactional mechanisms for implementing the semantics. 3) It describes a prototype implementation of the new techniques in the MobileBuddy system and provides measurements of the prototype supporting our performance claims.
The rest of the paper is organized as follows. Sec. 2 and Sec. 3 discuss the basic concepts, i.e. realization of escrow synchronization without modifications on the server side. Sec. 4 presents in more detail the transaction system that employs these ideas. Sec. 5 describes the split and transfer of escrow reservation without server intervention. Sec. 6 provides the MobileBuddy system implementation details. Sec. 7 presents the results of a client-side performance overhead study. Sec. 8 relates our approach to previous work. Sec. 9 summarizes our conclusions.

2 Our Approach

Our goal is to provide effective support for disconnected transactions using escrow synchronization to access shared objects in systems such as inventory control. Specifically, using the mobile sales example, we require:

1. Ability to acquire sales reservations so that a salesman can carry out sales transactions while disconnected and be sure the transactions will commit without conflicts.
2. A proper outcome in the absence of failures. For example, the salesman should be able to commit only the sales he ultimately manages to finalize.
3. A proper outcome in case of failure. For example if the salesman never finalizes the sale, the reservation should be released.

We have developed a new approach based on specialized escrow objects to support these requirements. Unlike earlier work, our approach requires no special processing on the storage server nodes. This is attractive because it allows to use generic commodity nodes. Earlier work also made use of specialized escrow data types to avoid concurrency control conflicts and developed a number of implementations 26, 20, 40, 19. However, all these approaches involved the use of specialized code running at the server node. Using our approach, these earlier escrow schemes can be adapted to use unmodified generic servers.

In our scheme, the persistent storage for objects resides on storage servers while mobile clients cache and access local copies of these objects. A disconnected client runs top-level disconnected transactions that contain within them special smaller revertible transactions (called in literature open nested transactions [25, 41]). The revertible transactions perform modifications to objects that are cached on a mobile client and are used to commit changes, e.g. reservations to items in stock, that may be cancelled later. They allow clients to coordinate their changes. Fig. 1 summarizes the steps taken by a mobile client both when connected and disconnected from the server.

Our approach to not run any special code at the storage nodes implies that storage nodes do not know anything about the revertible changes. Instead, storage nodes process all commit requests, including revertible transactions, identically. Our approach, instead, has special processing performed at the client machines. These computations run on cached copies of data from the storage server nodes, and these copies will reflect the changes made by other committed transactions, including both committed top-level transactions and committed revertible transactions. Thus the computations can observe the revertible changes of other disconnected transactions and take these into account.

Our approach makes use of special escrow objects. Such an object provides the normal operations, including obtaining or releasing a reservation for a resource. Additionally, these objects are prepared to handle the changes committed by revertible transactions. When the user calls a modification operation on such an object, the operation performs the modification and records the execution of the operation in a log along with a lease. The lease stores the time at which the revertible
Begin top-level transaction
Obtain reservations
Loop
    Refresh/load objects into local cache
    Disconnect from server
    Perform local revertible transactions
    Validate revertible transactions against reservations
    Record transaction results
    Reconnect to server
    Release some reservations if desired
    Renew reservations if necessary
    Obtain new reservations if desired
}
Commit top-level transaction i.e.
atomically validate/abort local transactions
and release unused reservations.

Fig. 1. Mobile client steps

operation will expire. The information about the revertible modifications and their leases is part of
the representation of the object. and thus is written to the storage server when the mobile client
reconnects and the application commits the revertible transaction. Other clients, upon connecting
to the shared storage server, will observe the revertible modification on the special escrow object.

When the client reconnects and is ready to commit the top-level transaction, it must first call a
special confirm operation on all escrow objects on which it wants the revertible change to become
permanent. This operation updates the status of that change so that it no longer appears revertible.
Additionally, the transaction can call a special release operation to undo the modifications that are
no longer of interest to it. Thus when the top-level transaction commits, all of the escrow objects
whose modifications have been confirmed will be stored with those changes having really happened,
and objects whose changes have been released will have those modifications cancelled. Note that the
application need not explicitly cancel `release`, the changes that are no longer needed, since these
modifications will be undone automatically when those objects are used by other transactions after
the leases expire. However, cancelling is desirable since it can release the resource earlier, before
their leases expire.

Fig. 2 depicts the structure of our system that supports applications that use escrow synchroniza-
tion (clients C1 and C2), without modifying the generic servers, or the applications that do not
use escrow (clients C3 and C4).

3 Exo-leasing

Consider the value of the shared object tracking the balance of in-stock items for sale in the
disconnected sales application, and consider the write/write conflicts that occur when concurrent
transactions add or remove item reservations. These conflicts are superfluous in the sense that,
as long as there remain available items for sale, no matter in what order the reservations are
interleaved they produce the same in-stock balance. A type-specific concurrency control scheme called escrow [26] avoids these unneeded conflicts by exploiting the semantics of the escrow type. An object of escrow type provides two commutative operations: split(delta) and merge(delta). A transaction calls the split operation to make a reservation for specified (delta, escrow amount), and calls the merge operation to return the unused escrow amount. As long as the in-stock balance is positive, the escrow locking protocols allows concurrent transactions to interleave the split and merge operations without conflicts. The escrow type is a representative of a general class of fragmentable objects [40]. Objects of this class have commutative operations that can be exploited by type-specific concurrency control schemes like escrow to avoid conflicts.

Escrow is a simple and effective synchronization method that has been well-known for a long time but has not been widely deployed in commercial systems. A principle barrier to the adoption in practice has been the need to modify the (legacy) concurrency engine since prior proposals for escrow synchronization code in the server.

We show how to perform escrow synchronization actions at the clients yet allow the same concurrent operation inter-leavings allowed by other escrow proposals. Our disconnected client/server system runs transactions on cached state in the client. Using a generic fine-grain read/write concurrency control scheme, and a cache coherence protocol that sends invalidation to the client if the object cached at the client becomes stale (because another client has modified it).

**Server side synchronization** Consider the server side implementation of a sales account service using escrow synchronization. The service is implemented by an object “service object” that exports a collection of methods. The methods include acquire, release, and expire operations that can be overridden by different fragmentable object implementations.

The object implementation consists of the procedures implementing the operations and the representation for the shared state they manipulate. The representation includes a set of outstanding
reservations and an internal in-stock balance object that implements the escrow operations. The split(delta) operation is called by the acquire request to obtain the reserved escrow amount, and the merge(delta) operation is called by the release request to return the unused escrow amount. The merge(delta) operation is also called by the expire method that is invoked internally by the service system when a reservation expires.

The reservation requests run as atomic transactions. The acquire request atomically commits the modifications to the in-stock-balance object and inserts a record describing the reservation into the reservation set. The reservation record specifies the reservation expiration time, and the actions that need to be performed if the reservation expires. These reconciler actions are type-specific, they perform the inverse of the operation invoked by the acquire request. The release and expire requests atomically commit the effects of the corresponding merge operation and remove the reservation.

The synchronization code described above resembles a concurrent object with a type-specific lock manager implemented using a monitor where monitor procedures implement the reservation requests, and monitor state encapsulates the internal in-stock-balance object and the outstanding reservation set. Within the monitor, the procedures use a simple mutex to serialize accesses to the shared monitor state.

**Client-side synchronization** A disconnected client/server system that runs transactions on cached state in the client, validates read/write conflicts at the server, using a cache coherence protocol that detects stale cache entries, can run the concurrent object on the client side. This is achieved by simply storing the persistent monitor state at the server, caching at the client the monitor code and state, running the monitor procedures on the cached state, and replacing the mutex synchronization with the cache coherence protocol that coordinates access to cached state by validating read/write conflicts at the server.

When the client is connected and issues a reservation acquire request, the corresponding monitor procedure updates the client’s cached state (the reservation set and the state of the in-stock-balance object) to reflect the reservation and sends the modified state to the server. If the state sent to the server is not stale, the server can commit the request making the updated state persistent. If the cached state is stale because another client has committed a reservation request, the server aborts the request and informs the client. The client obtains from the server the up-to-date monitor state, re-runs the request, and re-tries the commit with the new state. Eventually the request will succeed.

Consider a client that needs to return unused reservations. The commit of the release request is similar to the acquire request in that it may need to be retried.

**Lease expiration** Next consider the expire request. In the server-based scheme, the monitor code notices an expired reservation and invokes the expire request to release and reconcile the reservation. In the client-side scheme, the expired reservation is noticed by the monitor code at a client. Such a client obtains from the server the monitor state for the object, notices the expired reservation, and invokes the expire request to release the reservation. A reservation expiration may not be noticed for a long time if no client runs a reservation request. On the other hand, the expiration is of no interest until then. There is no problem with concurrent duplicate invocations of the expire request for the same expired reservation at multiple clients since after the first expire request commits other cached monitor copies become stale.

Detecting lease expiration at the client requires to make sure that a client with a fast clock does not expire the lease too soon. Our protocol uses the server time for lease expiration (assuming monotonic clocks). That is, a client must have received a message from the server with a timestamp greater than the lease expiration time.
We assume the server enforces object access controls so that only clients having suitable permissions are allowed to modify the monitor state. Since all escrow reservation requests require write permission the expired reservation can be reconciled by any client that makes a reservation request. Otherwise, the reservation reconciliation may need to wait until noticed by a client with appropriate permissions.

We call the above client-side concurrency approach exo-leasing (externalized leasing), and call the object running the escrow synchronization code at the client side as the escrow leasing object, or escrow object for short. We showed how exo-leasing works for escrow type. The same approach works for other fragmentable types [40]. A workshop position paper [34] by first author further considers a general transformation for deriving client-side type-specific synchronization schemes from server-side schemes.

**Considerations** Moving code to the client can adversely impact the performance of the system. The performance impact depends on the size of the monitor object that needs to be refreshed. The cost could become significant if the object is large and the contention is high. In general, however, we expect the synchronization objects to be small and contention levels to be moderate.

Moving code to the client raises a security concern if servers are trusted and clients and servers belong to different administrative domains. A rogue client could corrupt the monitor code, e.g. expire a lease “too early”, and commit changes that depend on the expiration request. Digital signatures can provide **accountability** [42], allowing to detect a rogue client after-the-fact, but may introduce overhead. The security concern is mitigated if a client runs in a secure appliance. A possible general solution is to exploit the TCB extensions proposed by the A2M (attested append-only memory) system [7] to provide “attested” escrow leasing objects. The details of the A2M-based approach are outside the scope of this paper and considered future work.

4 Disconnected transaction system

We have designed a 2-level transaction system that provides escrow synchronization for disconnected client transactions accessing shared objects stored in generic storage servers. Disconnected clients can validate transactions independently of other clients, and avoid, in the normal case, the after-the-fact conflict reconciliation. In the 2-level system, a generic base system, assumed as given, provides disconnected client/server storage for transactions accessing persistent objects. The base system synchronizes transactions using a read/write optimistic concurrency control scheme. Higher-level transactions, called application transactions, correspond to activities meaningful to the application. For example, reserving items for sale, running a disconnected sale, and then committing the sales transaction upon reconnection, may constitute one application transaction. Application transactions use base transactions to install durable updates. Application transactions synchronize using escrow objects.

This section specifies the base system and describes how we use it to implement escrow objects, to provide high-level transaction atomicity in the presence of client crashes and failures to reconnect, and to support disconnected high-level transaction validation. A technical report [36] further considers the formal correctness properties of the high-level transaction system, including semantic serializability.

**Base transactions** The base transaction system is based on the MX disconnected object storage system [35], though we could use any generic client-server storage system that supports cached
transactions, e.g. SQL server replication. A disconnected mobile client runs tentative transactions, accessing the local copies of the cached objects stored persistently in storage servers. A tentative transaction records intention to commit and allows the client to start up a next transaction. Tentative commits lead to dependent commits [14]: transaction $T_i$ depends on $T$, if it uses objects modified by $T_i$ because if $T_i$ ultimately aborts so must $T_i$. That is for tentative transactions $T_i$ and $T_j$:

$$T_i \text{ depends on } T \iff \text{ReadSet}(T_i) \cap \text{WriteSet}(T_i) \neq \emptyset$$

A tentative commit that is not a dependent commit, defines an independent action [9]: a transaction $T_i$ that does not use objects modified by $T_i$ can commit even if $T_i$ aborts.

To commit a tentative transaction persistently, the client connects to the server. An optimistic concurrency control scheme (adaptation of OCC [3]), provides efficient validation of disconnected client transaction read and write sets using invalidations. The server accumulates the invalidations for objects cached at a disconnected client, allowing upon client reconnection, to validate client transactions efficiently, including transactions accessing objects acquired from other clients while disconnected (using disconnected cooperative caching) [35]. A transaction that passes server validation is committed, and its results are stored persistently at the server (without re-executing it).

**Application transactions and escrow objects** Application transactions invoke operations on regular cached objects and encapsulated escrow objects. An application transaction runs as a top-level transaction that contains nested base transactions (tentative or durable). Application transaction effects become durable when it commits a base transaction at the server.

The escrow object operations (e.g. acquire, release and expire) run as base transactions nested inside the top-level transaction. They manipulate an escrow object representation consisting of regular cached objects. For example, the operation to acquire an escrow reservation that reserves a number of sales items reads the cached copy of the escrow variable to check if a sufficient amount of sales items is available for the reservation, and updates the cached representation to reflect an acquired amount. The base transaction that commits the acquire operation updates the durable copy of the escrow object at the server.

The nested transaction that commits an update to an escrow object at the server, without committing the top-level transaction, exposes the effects of the top-level transaction to other clients. Such open nesting [41, 25] allows to synchronize top-level transactions running in concurrent clients to avoid conflicts (e.g. another client can observe the existing reservations and reserve the remaining sales items).

Note, that since base transactions are optimistic, the server will abort a client base transaction if the cached escrow object state is stale, i.e. has been modified by another client. In such a case, the first client refetches the new state of the escrow object, re-executes the nested transaction on the fresh state, and retries the commit of the base transaction. The nested transaction is retried without undoing the top-level transaction.

**Atomicity** We want to guarantee the atomicity (all-or-nothing) property for top-level transactions. A top-level transaction that exposes its effects by committing open nested transactions (running escrow operations) can subsequently crash or abort. The exposed effects need to be undone (recovered) by running escrow operations that revert the effects. The protocol that accomplishes this resembles logical recovery for highly-concurrent data structures. e.g. ARIES recovery for indexes [12]. Likewise, its mechanisms, cleanups and reconcilers, resemble, respectively, logical recovery procedure and
logical undo records. Our protocol differs because it runs on the client side, and deals with leases rather than locks.

Cleanup runs when transactions commit or abort. The goal of the abort cleanup is to revert the exposed effects of an open nested transaction when the top-level transaction aborts. The goal of the commit cleanup is to ensure that the exposed effects are not reverted when the top-level transaction commits. The cleanup actions invoke operations called reconcilers, defined by the escrow objects. Reconcilers revert the effect of escrow operations. For example, the reconciler for an operation that acquires an escrow lease on an item, is an escrow merge operation that returns the item. The reconcilers are stored in the part of the escrow object representation, called the reconciler log. A reconciler is recorded in the log when the open nested transaction runs the associated escrow operation. A reconciler becomes durable when the open nested transaction commits at the server. The reconciler entry in the reconciler log can be active, deactivated, or timed-out. The open nested transaction commits an active reconciler that includes the lease expiration time.

An abort cleanup runs when a top-level transaction aborts. Abort cleanup invokes and deactivates the active reconcilers recorded by its open nested transactions. An abort cleanup can also run on a different client that has not created the reconciler but observes the timed-out reconciler in the reconciler log. Such abort cleanup runs (invoking and deactivating the reconcilers) when the top-level transaction at the observing client commits or aborts.

A commit cleanup runs when a top-level transaction commits. Commit cleanup deactivates the active reconcilers that have been recorded by its open nested transactions (without invoking them). The commit cleanup resembles the release of locks at transaction commit time in read/write locking schemes but there is an important difference. Where the release of read/write locks only affects performance, escrow leases must be removed (deactivated) atomically with the top-level commit to maintain correctness. This is because, if the top-level transaction commits and subsequently client crashes without removing the escrow leases, the time-out of the escrow lease will revert the effects
of the lease, thus violating the all-or-nothing property of the top-level transaction whose commit depends on the acquire of the lease.

A top-level transaction assembles the cleanup actions by registering callbacks to escrow object handlers called cleanup handlers when open nested transactions invoke escrow operations. In addition, validation procedures check the leases in the reconciler logs, and register handlers for the timed-out reconcilers so that commit or abort cleanup at the observing client will invoke the timed-out reconciler and deactivate it.

The cleanup action runs as a nested base transaction that commits (or aborts) atomically with the top-level transaction. To commit a nested transaction as part of the top-level transaction commit, the client simply includes the read/write sets of the nested transaction with the parent read/write sets. If the server can not commit the joint transaction because the escrow object was stale, the client receives an invalidation for the escrow object, refetches the new state of the object, and retries the joint commit without aborting the top-level transaction. Note that if other data (not the escrow object) was stale, the application will need to resort to after-the-fact reconciliation for that particular data.

**Example execution with escrow objects**  Fig. 3 shows the state of the reconciler logs at the server and two concurrent clients C1 and C2 running top-level transactions using escrow synchronization. The initial escrow object state at the server contains 15 in-stock escrow items and an empty reconciler log. C1 runs a top-level transaction T1 acquiring a reservation for 5 items by committing (step 1) a connected open nested transaction that updates at the server the escrow variable e to 16 items to reflect the remaining in-stock amount. Recording a leased reconciler merge 5/C1, that will unc its effect if C1 does not reconnect in time (lease time omitted to avoid clutter). Note, unlike for regular cached objects, after invoking acquire, the cached escrow amount at the client and at the server are different. A concurrent client C2 runs a top-level transaction T2 that acquires a reservation for 3 items (step 2) updating the durable escrow variable e to 7 to reflect the remaining in-stock amount, and recording a reconciler merge 3/C2. C1 consumes 2 escrow items while disconnected (running a special validated DON-transaction explained below that records tentative update to the cached escrow variable, resetting cached e to 3. If T1 were to abort at this point, the entire acquired amount has to be reconciled. If T1 were to commit, only the remaining uncommitted amount, as indicated by the cleanup handlers onCommit and onAbort registered with T1 (depicted within unshaded box). C1 reconnects and commits (step 3) the parent transaction T1, releasing the unused escrow amount and resetting the durable value of e to 10. The commit deactivates C1’s reconciler in the durable reconciler log (deactivated is entry marked *merge 3/ C1*). The durable reconciler log at the server still contains the active reconciler for the open nested transaction committed by C2, if C2 crashes, or does not reconnect in time, the reconciler will be invoked and deactivated by another client that accesses this escrow object and observes the expired reconciler.

**Semantic serializability** Top-level application transactions do not provide the read/write isolation property 11] because they expose uncommitted effects. Instead, top-level transactions are semantically serializable. We discuss this property informally, the formal treatment is outside the scope of this paper. Fig. 4 shows the semantic serialization of the two top-level transactions T1 and T2 run by C1 and C2 discussed above. assuming C2 reconnects in time and commits. The figure is intended to be read with time passing from left to right. At the low level we see the sequence of committed base transactions corresponding to the open nested transactions T1.1, T2.1, and the
nested transactions T1.2 and T2.2, issued by top-level transaction T1 'client C1' and T2 'C2), accessing the escrow object. In addition, T1 and T2 contain reads and writes of the regular objects (denoted r(Ti), w(Ti)). The nested transaction T1.2 [T2.2] runs when T1 (T2) commits atomically the effects of both escrow and regular object modifications. While in reality the reads and writes of T1 and T2 are interleaved, the semantic serializability of the T1 and T2 requires that it be as if the effects of T1 and T2 are isolated. Indeed, at the object level, since the split and merge operations commute, the committed order is equivalent (semantically) to T1.1, T1.2, T2.1, T2.2. Thus, T1 and T2, even though they are tangled together at the lower level of the base transactions, are semantically serializable top-level transactions.

Disconnected validation Our system supports disconnected validation [30] for top-level transactions. Disconnected validation guarantees, at the cost of extra checking at tentative commit time, that transactions will pass connected validation provided the client recontacts in time. Validated transactions are a useful practical abstraction that reflects the reality of disconnected computation. For example, a “guaranteed mobile sales transaction” that performs a disconnected update to the escrow object, runs as a validated transaction. A new variant of disconnected tentative transaction we call DON ‘disconnected open nested’ transaction supports disconnected validation. Unlike regular tentative transactions in the base transaction system, DON transactions run protected by the escrow lease and therefore can be validated by a disconnected client. Escrow operations run as DON transactions. A specialized validation procedure provided by escrow objects checks lease expiration. Our performance evaluation in Section 7 considers the overhead of the extra checking required to run DON transactions.
5 Lease Transfer

A disconnected salesman may want to transfer a reservation to a partner. Our system allows a
disconnected client (the requester) to acquire reservations from another client (the helper). Some
disconnected client/server systems allow one disconnected client to obtain consistent objects from
another client [5, 35, 28] but none support the transfer of reservations (locks or leases). Yet, such a
feature might be useful since it reduces the need to communicate with the servers and permits a
new pattern of collaboration within disconnected workgroups.

Example Consider how reservation transfer might be used in a scenario where a team of three
traveling salesmen: Joe, Sally, and Mary share a sales service account. Mary and Sally each obtain
a reservation (lease) to sell five items. Disconnect and travel together to a sales destination where
each completes a sales transaction selling one item each. Mary finds out she needs to change her
plans and leave for a different destination. Mary would like to transfer her remaining reservations to
Sally. This would allow Sally to guarantee the additional sales transactions she hopes to accomplish
to cover for Mary. Sally acquires the remaining four reservations from Mary, completes five sales
transactions and, before leaving for another destination, transfers her remaining reservations to
Joe who arrives to replace Mary. Sally reconnects to the server and successfully commits her sales
transactions, recording the reservation transfers. The commit reflects the sales of six reserved items,
removing the appropriate reservations for the sold items and adjusting the pending reservations to
four items. Mary reconnects next, recording a reservation transfer to Sally, and commits her sales
transaction. The commit reflects the sale of one reserved item, adjusting the pending reservations to
three items. Joe gets distracted with other matters and lets the remaining reservations expire. The
expire method is invoked, canceling the expired reservations and making the three reserved items
available again. At this point, to run a guaranteed sales transactions Joe would need to reconnect
and acquire new reservations. Fig. 5 summarizes the steps taken by a mobile client using reservation
transfer.

Exo-leasing makes reservation transfer possible because the escrow object that runs at the client
rather than the server, encapsulates the complete logic of the escrow reservation manager. The
reservation transfer is implemented by a special transfer procedure defined as part of the escrow
object implementation.

Fig. 6 depicts the structure of the system that allows one application to transfer escrow reserv-
ations between disconnected clients (clients C1 and C2), without modifying the generic servers or
other applications that do not use escrow synchronization but possibly use disconnected cooperative
caching provided by the base transaction system (clients C3 and C4).

5.1 Transfer Semantics

Escrow reservation transfer has to preserve the semantic invariants of the escrow type. Such transfer
should have the same effect as if the helper never had the reservation, and the requester acquired
the reservation by interacting with the server. That is, the transfer of a part of escrow reservation
from the helper to the requester must simultaneously increase the amount of escrow in the requester
and decrease by the same amount the reserved amount in the helper.

The correctness condition for reservation transfer [36] requires that a transaction system that
commits transactions using the transferred reservations is equivalent to a system that commits the
same transactions without the transfer, where all reservations are obtained by interacting with the
**Fig. 5.** Client steps with reservation transfer

**Fig. 6.** Escrow Reservation Transfer between Clients

server. Of course, any one of the disconnected clients participating in the typed lease transfer can
crash before reconnecting to the server. Moreover, the participating clients can reconnect in any order. For example, a requester that has acquired the reservation from a helper could reconnect first, and the helper that has supplied the reservation could crash while disconnected. The correctness condition for reservation needs to be maintained in the presence of disconnected client crashes and all possible participant reconnection orders.

5.2 Transfer Atomicity

We implement the reservation transfer procedure using a new kind of transaction. The new transaction, called a 2DON transaction, involves two clients participating in the transfer. Each client runs a nested tentative base transaction. 2DON transaction is tentative because one or both participants in the transfer could crash. It commits durably when one or both participants reconnect to commit the transaction at the server.

To ensure the atomicity of the transfer, the 2DON transaction has to record enough information in the participants to enable any reconnecting participant to recover independently, if the other participant does not reconnect in time. The reservation transfer procedure at the helper client calls the

\texttt{move} object release operation to reflect the transfer. This updates the cached \texttt{move} variable and defines the appropriate commit and abort cleanup actions, recording the appropriate reconcilers in the reconciler log. These steps are identical to DON transaction. In addition, the reservation transfer procedure records the reconcilers of the other participant so that the reconciler logs at both participants contain identical sets of reconcilers reflecting the transfer. Using the reconciler logs the eventual cleanup actions ensure that reconcilers for unused reservations are invoked (and deactivated), and the reconcilers for used reservations are deactivated.

In a client that reconnects first, the commit cleanup for the top-level transaction containing the transfer deactivates the durable original reconciler created when the reservation that got transferred was first acquired, and adds the two reconcilers, generated by the transfer. The reconciler for the amount held by the reconnecting committing client gets immediately deactivated when this reconnecting transaction commits. The reconcile for the amount held at the other participant will get deactivated when the second participant in the transfer reconnects (or by expiration).

Our protocol guarantees that the appropriate cleanup actions for every reconciler will be invoked exactly once, in all three cases that constitute the possible outcomes of the transfer: when the second participant reconnects and commits in time, second participant fails to reconnect, or none of the participants reconnect in time. An example illustrates the protocol steps.

\textbf{Example execution with transfer} Fig. 7 depicts the reconciler logs reflecting escrow reservation transfer. The execution steps are identical to the example in Figure 3, except for lease transfer in step 2. C1 consumes 2 units and resets the cached escrow object to 3, running a validated DON transaction. Then, C1 encounters another client and, as a helper, transfers a reservation for 3 items to a disconnected requester C2 (step 2). The transfer procedure runs as the 2DON transaction, resetting the escrow amount to 1 in the helper, and to 3 in the requester. The 2DON transaction records in the reconciler logs of the participants the leased reconcilers reflecting the transfer (same expiration time as the original helper lease). The requester transfer reconciles (merge 3/C2 accounts for the case when the requester does not reconnect in time. Such a lost transferred amount needs to be recovered by merging it back into the total available amount. The reconnecting helper C1 that has participated in the transfer, and therefore contains this reconciler, will durably commit it at the server. The expiration of this reconciler will trigger the intended cleanup. This reconciler will
be deactivated by the requester if it reconnects in time as part of the commit of the top-level transaction that ran the 2DON transaction.

The helper transfer reconcile split 3/C1 accounts for the situation where the helper does not reconnect in time and therefore does not deactivate the durable reconcile merge 5/C1 generated by C1’s open nested transaction that obtained the original reservation. The timeout of the original acquire reconcile could (incorrectly) merge back the entire amount not accounting for the transferred amount. The reconnecting requester C2 participated in the transfer, and therefore contains the reconcile split 3/C1 generated by the 2DON transaction. Will durably commit it at the server. The expiration of this reconcile, together with the original acquire reconcile will trigger the execution of both reconciles, resulting in the cleanup that correctly adds back 2 escrow units. Both reconciles, corresponding to acquire and transfer, would be deactivated by the helper if it reconnects on time as part of committing the top-level transaction T1.

In the example, the requester reconnects to the server first (step 3), releasing all the reservations (assume no reservations were used up), updating the server escrow amount to 13 to reflect the returned escrow. Deactivating the requester’s transfer reconcile, and updating the server reconcile log to include the helper’s reconcile. All of the above actions are committed atomically by a base transaction. After the commit, the cached requester state contains no reservations (not shown).

Consider the possible ways the execution could proceed. If the helper reconnects and commits on-time, this would not change the durable escrow value but would deactivate the helper’s reservations and corresponding reconciles, as explained above. Alternatively, if the helper does not reconnect, the reconcile log at the server containing the helper’s reconciles merge 5/C1 and split 3/C1 would eventually be observed and invoked at some other client adding back 2 units. If both the requester and helper fail to reconnect in time, some client eventually accessing the escrow object would invoke the timed out reconcile |merge 5/C1| stored originally in the reconcile log.
6  The MobileBuddy System

We implemented MobileBuddy, a 2-level transaction system providing application transaction system on top of the MX disconnected object storage system [35].

**MX** The MX system supports connected servers and mobile collaborative client groups. Servers provide reliable and highly-available disk storage for the master versions of persistent objects. A persistent object is owned by a single server. Since objects may be small (order of 100 bytes for programming language objects [23]), objects on disk are clustered into physical pages. Less reliable mobile devices run client applications that access objects on cached copies of pages. Mobile clients either run disconnected from the servers, or connect through a high-latency WAN. The client population is dynamic but not changing fast since access to the shared persistent storage system requires an authenticated account. Mobile clients have efficient means to communicate with each other and can dynamically form collaborating groups, interacting at a close range. A typical group size does not exceed 5-10 clients; smaller groups being the common case. Before disconnecting from the servers, a mobile client pre-loads the cache with up-to-date copies of the master versions of the objects of interest.

A disconnected client can transfer pages to other clients using disconnected cooperative caching. While disconnected, a mobile client runs tentative transactions accessing the local copies of the cached objects. At any time the mobile client has access to two cached data versions: a local version in the working cache, and a best-known master version in the clean cache. A tentative transaction updates the working 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, and undo records. The undo records support low-cost fine grain roll-back to the pre-transaction state. The best-known master versions are used in the disconnected transfer of pages ‘called peer-fetching,’ between clients. A client peer-fetching a page from another client, re-validates its tentative transactions if the fetched page has more recent objects of interest. A mobile client reconnects to the servers to commit the tentative transactions ‘reporting peer-fetched pages if any. The server validates the transactions using accumulated invalidations and creates new durable versions of the modified master copies.

**MobileBuddy** The MobileBuddy system implements the application transaction and exo-leasing protocols described in Sec. 4 and Sec. 5. To support expressive applications, following Mobisnap [30], in addition to escrow leases, MobileBuddy provides a set of additional generic leases corresponding to the locking modes supported by SQL systems. We support value-change, value-use, and shared-value-change leases, implemented using exo-leasing, and provide lease transfer for value-change and shared-value-change leases. The complete leasing system description (omitted for brevity) appears in 39.

The MobileBuddy system integrates with MX client side code. The escrow objects are implemented as persistent MX objects. Escrow objects provide the acquire, release and expire operations, and contain as part of their representation the escrow variable and the internal reconciler log. In addition, escrow objects provide the commit and abort cleanup handlers and a reservation transfer procedure. The transfer procedure is integrated with the MX peer fetch protocol. The cleanup actions associated with the open nested transactions (DON and 2DON), are implemented as callbacks (to escrow object handlers) invoked by the MX transaction manager. In addition, the escrow object provides a procedure for the disconnected DON transaction validation, checking the escrow lease
validity invoked by the generic disconnected validation procedure (checking the generic types of
leases mentioned above).

We modified the MX client side protocol to support leased objects and nested transaction
commit as follows. When the client reconnects and receives invalidations, the invalidation received
for leased objects are intercepted and ignored as long the lease is valid and the client does not
attempt to update the escrow object, i.e., client retains the leases. Escrow object invalidations
received during commit of a transaction that updates escrow objects causes the the client to
refetch the new escrow object state. MobileBuddy re-runs the escrow operation on the new state
invoking cleanup, and attempts to commit again. This approach is used both for the open
nested transactions that acquire escrow reservations and the nested transactions that run during top-level
transaction commit with cleanup.

6.1 Supported leases

Different types of objects require different kinds of leases. MobileBuddy supports two basic classes of
leasing objects: generic and type-specific. The generic leases include value-change (vc), value-use
(vu), and sharedvaluechange (shv) leases corresponding to the locking modes supported by SQL
systems. The type-specific leases support the fragmentable escrow objects (denoted e). Additional
types of leases may be needed for new types of application data, but these four are typical and
they support a rich set of common applications [30]. Typed leasing objects for additional types are
described in [39].

The implemented lease semantics are as follows:

\texttt{vc}: gives one client the exclusive right to change the object it leased, similar to a write lease.
\texttt{vu}: provides the right to use a given value for some data item despite its current value at the server.
\texttt{shv}: provides the guarantee that it is possible to modify the state of an existing data item at the
server. It prevents other clients from obtaining a \texttt{vc} lease on the same object. For a group of
collaborators who hold the \texttt{shv} it serves as a group lease that guarantees whichever collaborator
reconnects first will be able to commit modifications.
\texttt{e}: provides the exclusive right to use a specified share of a partitionable resource represented by
a fragmentable object [40], often a numerical data value. It supports two operations \texttt{split} and
\texttt{merge}. A split operation on an object \( o \) removes a requested amount \( \text{delta} \) from the object and
returns \( \text{delta} \). If the requested amount is unavailable, the operation signals exception and has
no effect. The merge \( \text{delta} \) operation on object \( o \) adds the specified amount \( \text{delta} \) to object \( o \).

The \texttt{vc}, \texttt{vu} and \texttt{shv} leases apply to all types of data while \texttt{e} reservations apply only to fragmentable objects. Different types of leases may be used in the same typed leasing object but not
all types of leases are compatible. Consider a typed leasing object providing all four lease types.
The compatibility rules (used in Mobisnap system [30]) are enforced by the leasing object imple-
mentation following the compatibility table 1. A \texttt{yes} entry means two types of leases can exist on
the same object at the same time. For an escrow lease \( e \), we also need to check if there is enough
value left to reserve.

Compatible leases enable clients to read/write the same object independently without creating
conflicts. Non-compatible typed leases, working like traditional locks, guarantee the result for the
client who holds the lease. The typed leases allow a mobile disconnected client to validate and
guarantee the final result of a tentative transaction provided the transaction only reads leased values
and the mobile client reconnects to validate the transaction before the leases expire. Therefore a
successf}ully validated disconnected transaction that has the appropriate leases on all objects it accesses is not affected by concurrent transactions except in the case of shvc leases because shvc does not provide a guarantee on the value of the leased object.

Acquiring an v lease on an object u changes the value of the object at the server, and thus the optimistic concurrency control protocol [35] generates invalidations for all the cached object copies. This does not invalidate other exo-leased copies of the object because the client side exo-leasing code ignores such invalidations.

In case the mobile client does not reconnect before a lease expires, the expired lease needs to be released. In the case of an expired v or vu lease, the leased object becomes invalid at the client. In addition, in the case of an e expiration, type-specific actions need to be performed, e.g. the leased amount must be made available for others to use.

Not all kinds of leases can be transferred. We do not support transfer of a vu lease because this kind of lease may reflect a certain value that is likely to be specific to the particular client who obtains such a lease. It would be easy to remove this restriction if in the future we find this type of exchange to be useful.

Lease transfer preserves the semantics of the lease. The effects of the transfer at the helper and the requester are as follows.

1. In a transfer of a vc, the helper hands over the lease to the requester. The helper does not hold the vu anymore.
2. In a peer-fetch of an e on object u, the helper splits value e off its own value of u. The requester adds the lease to its lease list and setting the value of e in its working cache to v.
3. In a peer-fetch of shvc, the requester adds the lease to its lease list, the helper keeps its lease.

7 Performance Evaluation Study

Our goal is to evaluate the performance of validated DON-transactions enabled by v and lease transfer the novel features introduced by our system.

Experimental methodology and findings Leases provides two types of benefits for disconnected collaborators. First, the ability to obtain leases and to run validated transactions while disconnected avoids loss of work due to conflicts and eliminates in the normal case some of the potentially high (but not entirely avoidable) costs of external compensation actions. This benefit, determined by the transaction workload and other application-specific costs, reduces to the general benefit of type-specific and generic locking, and has been studied before (e.g., the results in [30] apply to our system). Second, obtaining a lease from a nearby collaborator, instead of obtaining it by interacting directly with the server is advantageous in a weakly connected environment when the cost of communicating with the server is high. This benefit reduces to the benefit of disconnected cooperative
caching and has been studied before (e.g. the results in MX system [35], and others 5,28 apply). Our study does not repeat the evaluation of the known benefits of escrow leases and lease transfer. Instead, we evaluate the overhead introduced by validated DON transactions.

To gauge the overhead, we consider two possible situations: In one case, MobileBuddy transactions run while holding leases for all the objects they access and therefore can benefit from the validation. In the other case, transactions run with insufficient leases. Our experimental findings, using a standard benchmark, indicate that in a mobile transactional object system (many small objects), the extra overhead imposed by enabling validated DON transactions can be high for application transactions that do not benefit from the leases (i.e., fail validation, or do not need to be validated). As expected, the overhead for lease transfer is offset by the cost of accessing the server when network latency is non-negligible. Note, that MobileBuddy system incurs no additional overhead if the client holds no leases.

**Experimental Configuration** We run MobileBuddy in a system configuration where a server and the clients ran each on a 850MHz Intel Pentium III processor based PC. 256MB of memory, and Linux Red Hat 9.0, an obsolete version compatible with the aging MX system implementation. The experiments ran in an isolated system in the Utah experimental bested emulab.net [1] that enables access to older operating systems versions on a dedicated system. The cost of the leases is independent of the size of the collaborative group, given the small group sizes expected in MobileBuddy, and given we do not expect high lease contention. A system configuration containing a server and two clients is sufficient therefore for our experiment. All reported experimental measurements are averages of three trial runs showing minimal variance with hot caches.

**The OO7 Benchmark** Our workloads are based on the multi-user OO7 benchmark [6]; 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 us to control the sharing of complex data and because it is a standard benchmark for measuring object storage system performance. To study the cost of leases, we extended the OO7 database to support escrow objects. Now each atomic part has two additional escrow objects, so the application can acquire leases on the escrow objects. Otherwise, the database is the same as a normal OO7 database.

The cost of the leases is workload-dependent, proportional to the number of objects accessed by a transaction. In the extended OO7 benchmark, each transaction accesses 72,182 objects.

**Overheads** We evaluate two kinds of overhead incurred by validated transactions, the Penalty for transactions that do not benefit from leases, and the Cost for transactions that do. The two overheads differ because they occur under different circumstance, as we explain below. The Penalty is our main concern. It is due to the checks performed by the transaction validation system at the client to determine whether a transaction uses an object while holding a lease. The Cost is the overhead due to processing and transferring the leases for validated transactions that use the transferred leases, i.e., access objects protected by it. Our concern here is whether the checking cost is reasonable.

The validation overhead occurs at three points:

1. Tentative commit: each one of the objects accessed in the tentative transaction (the read set) is checked whether it is protected by a lease, to determine whether the tentative transaction (and its updates) can pass disconnected validation.
2. Transfer all tentative transactions that have accessed objects without leases before lease transfer are re-validated using the acquired leases.
3. Durable commit at reconnect: the client runs cleanup handlers registered by transactions using escrow objects.

**Penalty** Consider a mobile sales scenario discussed in Section 5. Suppose a salesperson Mary disconnects with leases, but her tentative transactions use objects unprotected by the leases. Assuming Mary enables the disconnected validation checks, each time Mary commits a tentative transaction, the transaction is validated introducing penalty *Tentative*, defined as the time of the check relative to the total tentative commit time. This cost is 9% in our experiment, but is workload dependent and is higher when the violation is detected later in the check since the check stops when violation is detected. For example, in the worst case in our workload, if all 72,182 objects are checked, this penalty adds 62ms per tentative commit.

In our scenario, when Mary meets with John, she further obtains some leases from John. Since her tentatively committed transactions have not used objects protected by leases, the transfer causes the validation of all her tentative transactions against the transferred leases resulting in penalty *Transfer* (*Tentative* per transaction). This cost would be offset by the cost of fetching leases from the server when the network latency is non-negligible.

When Mary reconnects to the server, the transaction commit protocol checks invalidations and runs cleanup handlers that update the persistent copies of escrow objects, removing leases and returning the unused amounts. We conservatively consider the worst case when Mary has obtained leases on all escrow objects, and all her escrow objects have pending invalidations due to John’s reservations. In this case, the client-side commit penalty *DurableCommit* is 32%, including *InvalidationChecks*, adding 7% extra relative to the total reconnection validation time, and *CleanupHandlers*, adding 25% extra to total validation time, adding extra 305ms to the total reconnection validation time. A realistic workload is unlikely to have that many escrow leases so the overhead will be lower.

**Cost** In this case, Mary disconnect with leases that are now used by her tentative transactions. Mary’s disconnected validation succeeds each time, but to detect this, she performs the validation at each commit checking all the objects that the transaction has accessed. This introduces the cost *Tentative*, defined as the time to validate relative to the total tentative commit time. This overhead is high, 47% in our experiment. Recall the difference between this overhead and the corresponding penalty is that when Mary does not use leases, the checking procedure stops when it finds the first unprotected access in Mary’s tentative transaction read set. In contrast, when she has enough leases, the procedure checks the entire read set.

Table 2 summarizes the client-side overheads of validated DON transactions.

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>TentativeCommit</td>
<td>47%</td>
<td>9%</td>
</tr>
<tr>
<td>Transfer</td>
<td>-</td>
<td>TentativeCommit * number of transactions</td>
</tr>
<tr>
<td>DurableCommit</td>
<td>32%</td>
<td>32%</td>
</tr>
</tbody>
</table>

*Table 2. Overheads of validated DON transactions*
There are two things to notice. First, recall the *Penalty* for lease transfer is incurred for each tentative transaction accessing objects without holding leases. There is no corresponding validation *Cost* associated with lease transfer since in this case transactions committed before the transfer have accessed objects while holding leases. Second, the *Penalty and Cost overheads for DurableCommit* are equal. Whether client uses a lease, or not, the connected durable commit cleanup actions check the invalidations and remove the lease, returning unused escrow amount.

Note, the client-side *DurableCommit* overhead is also incurred to obtain the leases before disconnection. The server-side overhead of obtaining and removing a lease is simply the cost of an update transaction.

**Summary** If the client obtains escrow leases but does not use them, the penalty of validated DON transactions are non-negligible. If the client relies on the reservations, using them to achieve disconnected validated transactions, then the client pays for the benefit brought by the reservations. We consider the cost reasonable.

8 Related work

Our work blends together a number of ideas. a disconnected client/server system, cooperative caching, escrow synchronization and multi-level transactions into a system that attacks a specific portion of the disconnected system design space, combining conflict avoidance and optimistic synchronization. To our best knowledge, none of the prior work has considered moving the synchronization out of the server, or disconnected client-to-client synchronization.

Most disconnected client/server systems use optimistic concurrency control and handle conflicts after-the-fact. A survey of mobile concurrency approaches can be found in [17]. The Coda disconnected file system [18] introduced server-side automatic resolution procedure for conflicting directory updates. The sprocket mechanism [29] provides safe server-side concurrency extensions. The Coda system allows applications to run on the client application-specific resolvers [ASR] for conflicting file updates [21], as does the Ficus system [32]. The difference between ASR and exo-leasing is that exo-leasing avoids conflicts (in the normal case) by coordinating in advance, supporting disconnected validation. A useful ability when the cost of after-the-fact reconciliation is high.

The MX system [35] introduced cooperative caching [8] for disconnected client/server system, allowing to transfer consistent objects from one client to another without contacting a server. MobileBuddy builds on MX transactions. Disconnected cooperative caching has been used in Ensemble [28] mobile appliance system. PRACTI 5 replication framework and work by Sailhan et al [33]. Most peer-to-peer storage systems that transfer mutable objects among peers support weak consistency. Lazy Replication [22] and Bayou [38] provide strong consistency for objects and allow type-specific update merging procedures. The mobile epidemic quorum system [16] provides multi-object transactions.

Despite a large body of literature on multi-level transactions and escrow that appeared in the mid 80's (most relevant approaches identified below), no commercial systems or applications that we know about have deployed these techniques. The need to modify the concurrency engine in the server has been the principal barrier. Weikum [41] introduced multi-level client/server transactions with open nesting in a database system with locking. Lomet [24] investigated general multi-level recovery. Our multi-level concurrency and recovery system differs from the earlier approaches because it is optimistic and deals with leases [10] rather than locks. Manon et al [25] describe an
optimistic approach using open nesting in transactional memory system [15]. The middleware implementation 31 of the J2EE Activity Service [2] increases concurrency for long-running connected transactions in a multi-level transaction system using semantic locks 12, as does the promises system 13. Our workshop position paper 34 discusses how to achieve a similar benefit for long-running transactions and snapshot queries using exo-leasing in a general type-specific concurrency scheme 37. The CheeTah middleware system 27 generalizes multi-level client/server transactions to a peer-to-peer system, providing on each peer server a lightweight transaction monitor.

Escrow synchronization was introduced by O’Neil 26 and extended to replicated systems by Kumar and Stonebraker 20. Walborn et al 40 generalizes escrow synchronization to fragmentable and reorderable data types. Krishnakumar and Jain 19 explore the use of escrow to improve throughput for mobile applications. The demarcation protocol 41 enforces semantic constraints to a similar effect. The approach in Mobisnap 30 mobile client/server storage system is closest to ours and has inspired our work. Like exo-leasing, Mobisnap combines optimistic concurrency with lease-based conflict avoidance and supports disconnected validation. However, like other proposals, Mobisnap implements the type-specific synchronization at the server.

9 Conclusion

This paper attacks an insufficiently studied problem in the mobile computing space, namely, how to support escrow synchronization in practical disconnected client/server storage systems so that disconnected clients can operate independently on shared data and validate transactions to avoid conflicting updates that later need to be aborted or reconciled. To that effect, this paper makes the following contributions: 1) It describes exo-leasing, a new modular approach to escrow synchronization that avoids type-specific code at the server providing the ability to use commodity servers.

2) It describes a reservation transfer mechanism that can aid collaboration in disconnected groups and is enabled by exo-leasing. 3) It presents MobileBuddy, a prototype escrow synchronization and reservation transfer system based on exo-leasing, and provides measurements of the prototype evaluating the client-side overhead of running disconnected validated transactions.

References

1. 'emulab.net', the Utah Network Emulation Facility. supported by NSF grant ANI-00-82493.


