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

Liuba Shrira[1], Hong Tian[2], and Doug Terry[3]

[1] Brandeis University
[2] Amazon.com
[3] Microsoft Research

Abstract. Escrow reservations is a well-known synchronization technique, useful for inventory control, that avoids conflicts by taking into account the semantics of *fragmentable* object types. Unfortunately, current escrow techniques cannot be used on generic "commodity" servers because they require the servers to run the type-specific synchronization code. This is a severe limitation for systems that require application-specific synchronization but need to rely on generic components.
Our *exo-leasing* method provides a new way to implement escrow synchronization without running any type-specific code in the servers. Instead, escrow synchronization code runs in the client providing the ability to use commodity servers. Running synchronization code in the client provides an additional benefit. Unlike any other system, our system allows a disconnected client to obtain escrow reservation from another disconnected client, reducing the need to coordinate with the servers. Measurements of a prototype indicate that our approach provides escrow-based conflict avoidance at moderate performance overhead.

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 [11]. 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

* This research was partially supported by NSF grant CNS-0427408 and Microsoft Research, Cambridge, UK. The work was done while Hong Tian was at Brandeis University.

at the storage server. If a conflict is detected the client has to abort the changes or reconcile them, possibly using application-specific resolvers [11, 21].

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 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 or unhappy customers. To avoid costly conflicts, a mobile client, before disconnecting, can obtain reservations [20](locks) that guarantee (in-advance) the successful completion of specific transactions while disconnected. *Escrow* synchronization [17] is a well-known simple scheme, useful for inventory control, that provides such reservations. It exploits the properties of *fragmentable* [29] object types to avoid conflicts when clients make concurrent changes to shared objects [20]. 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 paper is to fix this limitation. We describe a new technique called *exo-leasing* that provides escrow without running type-specific code at the servers. Instead, this code runs in the client. 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 exo-leasing is the ability to provide new functionality. Unlike any other system, we allow a disconnected mobile salesman on a sales trip to split and transfer part of a reservation to a partner. Exo-leasing makes this possible because the synchronization code running at the client encapsulates the complete synchronization logic. Disconnected reservation split and transfer reduces the need to communicate with the servers, providing a complementary benefit to disconnected cooperative caching [24, 18] that transfers data but not reservations.

We prototyped MobileBuddy, an exo-leasing escrow synchronization system on top of a generic transaction system, 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 server can support

escrow synchronization so that 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 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 a novel reservation split and transfer facility enabled by exo-leasing, describing its semantics and new transactional mechanisms for implementing the semantics. 3) It provides measurements of a prototype system, supporting our performance claims.

2 Our Approach

Our goal is to provide effective support for disconnected client transactions using escrow synchronization 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.

Our new approach, based on specialized escrow objects, supports these requirements, and, unlike prior work, requires no special processing on the storage server nodes. This is attractive because one can use generic commodity nodes. Prior work also made use of specialized escrow data types to avoid concurrency conflicts and developed a number of implementations [17, 12, 29]. However, these approaches involved the use of specialized code running at the server node. Using our approach, prior 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* (*open nested* [30]) transactions. 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 (for now ignore the split and transfer steps that will be explained later).

Our requirement 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

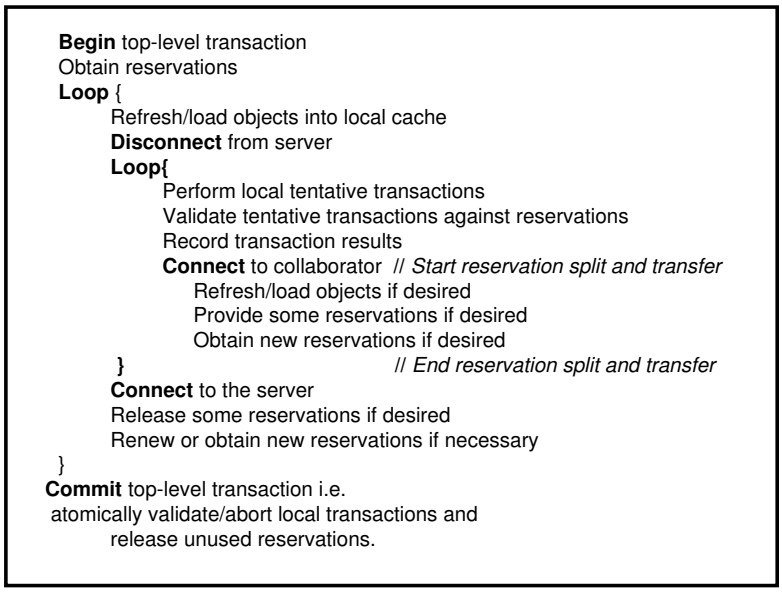

Fig. 1. Mobile client steps

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 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.

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 synchronization scheme called *escrow* [17] 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 [29]. Objects of this class have commutative operations that can be exploited by type-specific synchronization 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 run escrow synchronization code in the server. We show how to implement escrow 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 escrow The server-side implementation of a sales account service using escrow synchronization consists of 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 of 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 recovery 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 escrow 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, the server aborts the request and informs the client. The client gets from the server the up-to-date monitor state, re-runs the request, and tries to commit with the new state. Eventually the request will succeed. If a client needs to return unused reservation the commit of the release request is similar to the acquire request in that it may need to be retried.

Lease expiration In the server-side scheme, the monitor code notices an expired reservation and invokes the expire request to release the reservation. In the client-side scheme, the monitor code at a client notices the expired reservation. Such a client invokes the expire request to release the reservation. There is no problem with concurrent duplicate invocations of the expire request for the same expired reservation at multiple clients since after the first release commits other

cached monitor copies become stale. 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. 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. Note, that a client with a fast clock could expire the lease too soon. To avoid this, we use 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 call the above client-side synchronization approach *exo-leasing* (externalized leasing), and refer to the object running the escrow synchronization code at the client simply as *escrow object*. We showed how exo-leasing works for escrow type. The same approach works for other fragmentable types [29]. A general transformation from a server-side type-specific synchronization scheme to a client-side scheme is described in [23].

Considerations Moving code to the client can adversely impact the performance of the system if the monitor 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 could allow 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 approach, considered future work, is to exploit recently introduced hardware TCB extensions.

4 2-level Transactions with Exo-leasing

We have designed a 2-level transaction system that supports escrow synchronization for disconnected client transactions accessing shared objects stored in generic storage servers. In the 2-level system, a generic *base* transaction system, assumed as given, provides disconnected client/server storage for persistent objects. The base transactions synchronize using *read/write* optimistic concurrency control. 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 synchronize using escrow objects. We describe how we use the base system to implement escrow objects, to provide application transaction atomicity in the presence of client crashes and failures to reconnect, and to support disconnected

application transaction validation. A technical report [25] considers the ACID properties in our 2-level system.

Base transactions MX disconnected object storage system [24] provides base transactions, though we could use other 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* [9]: transaction T_j *depends* on T_i if it uses objects modified by T_i because if T_i ultimately aborts so must T_j. A tentative commit that is not a dependent commit, defines an *independent action*: [5] a transaction T_j 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 [2]), 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 reconnection, to validate client transactions efficiently, including transactions accessing objects acquired from other clients while disconnected (using disconnected cooperative caching [24]). 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 [30, 16] 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 re-fetches the new state of the escrow object, re-executes the nested transaction on the fresh

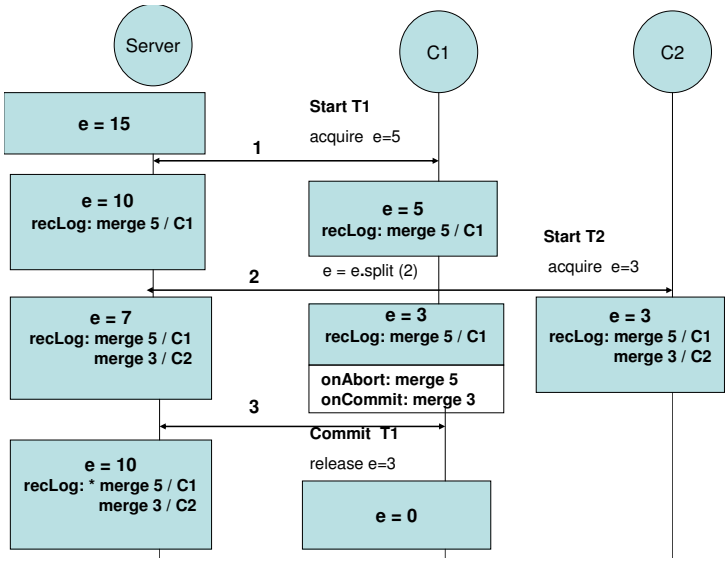

Fig. 2. Reconciler logs in escrow leasing

state, and retries the commit of the base transaction. The nested transaction is retried without undoing the top-level transaction.

Recovery 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 [7]. Likewise, its mechanisms, *cleanup* 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, running when a top-level transaction aborts, invokes and deactivates the *active* reconcilers recorded by its open nested transactions. An abort cleanup can also run on a different client that observes a *timed-out* reconciler in the reconciler log. Such abort cleanup runs when the top-level transaction at the observing client commits or aborts. A commit cleanup, running when a top-level transaction commits, 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. A 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, re-fetches 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 Fig. 2 shows the state of the reconciler logs at the server and two concurrent clients C1 and C2 running top-level transactions using an escrow object. 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 10 items to reflect the remaining in-stock amount, recording a leased reconciler [merge 5/C1] that will undo 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 unconsumed 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.

Disconnected validation Our system supports *disconnected validation* [20] 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 reconnects 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 runs as DON transactions. A specialized validation procedure provided by escrow objects checks lease expiration. Sec. 6 considers the performance overhead of disconnected validation.

5 Reservation Split and 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 [3, 24, 18] but none support the split and 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. Consider how reservation split and transfer might be used in a scenario where a team of three traveling salesman Joe, Sally and Mary share a sales service account. Mary and Sally each obtain a reservation to sell five items, disconnect and travel together to a sales destination where each completes a sales transaction selling one item each. Mary changes her plans, departing 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 departing, 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. 1 summarizes the steps taken by a mobile client using reservation transfer. Exoleasing makes reservation split and 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.

Semantics Escrow reservation split and 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 split and transfer [25] 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 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.

Recovery We implement the reservation split and transfer 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 insure 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 escrow object release operation to reflect the transfer. This updates the cached escrow 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 insure 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 reconciler 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.

Example execution Fig. 3 depicts the reconciler logs reflecting escrow reservation transfer. The execution steps are identical to the example in Figure 2, except for step 2 when helper client C1 connects to requester client C2 and splits and transfers a reservation for 3 items. The split and transfer runs as the 2DON transaction, resetting the escrow amount to 0 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 reconciler [merge 3/C2] accounts for the case when the requester does not reconnect in time. Such lost transferred amount needs to be recovered by merging it back into the total available amount. The reconnecting helper C1 will durably commit this reconciler at the server. The expiration of this reconciler will trigger the intended cleanup. This reconciler will be deactivated by the requester C2 if it reconnects in time as part of the commit of the top-level transaction T2 that run the 2DON transaction.

The helper transfer reconciler [split 3/C1] accounts for the situation where the helper does not reconnect in time and therefore does not deactivate the durable reconciler [merge 5/C1] generated by C1's open nested transaction that obtained the original reservation. The timeout of the original acquire reconciler could (incorrectly) merge back the entire amount not accounting for the transferred amount. This is not a problem because the reconnecting C2 has the reconciler [split 3/C1] generated by the 2DON transaction, and will durably commit it at the server. The expiration of this reconciler, together with the original acquire reconciler will trigger the execution of both reconcilers during cleanup,

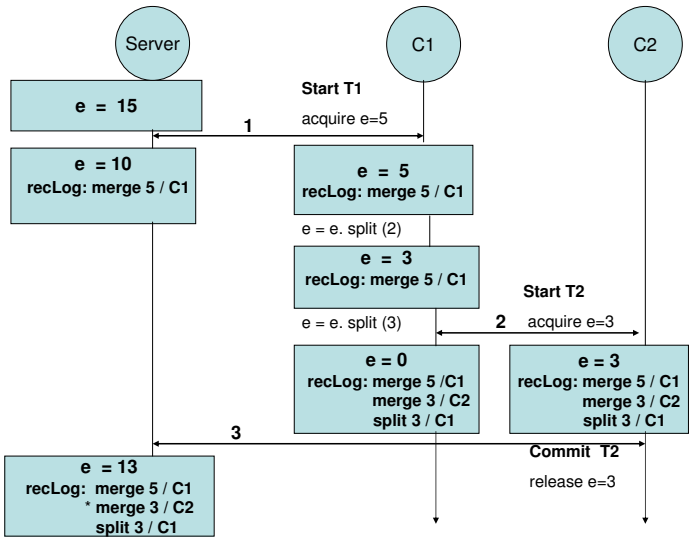

Fig. 3. Reconciler logs in escrow split and transfer

correctly adding back 2 escrow units. Both reconcilers (corresponding to acquire and transfer) would be deactivated if C1 reconnects on time as part of the top-level transaction T1 commit. In the example, C2 reconnects to the server first (step 3), releases all the reservations (no reservations were used up), updates the server escrow amount to 13 to reflect the returned escrow, deactivates the requester's transfer reconciler, and updates the server reconciler log to include the helper's reconciler. All of the above actions commit atomically in a base transaction. After the commit, the cached requester state contains no reservations (not shown). Consider the possible ways the execution could proceed. If C1 reconnects and commits on-time, this would not change the durable escrow value but would deactivate the helper's reservations and corresponding reconcilers, as explained above. Alternatively, if C1 does not reconnect, the reconciler log at the server containing the helper's reconcilers [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 reconciler [merge 5/C1] stored originally in the reconciler log.

6 Experimental Evaluation

We have implemented MobileBuddy, a prototype 2-level transaction system with exo-leasing and evaluated its performance. MobileBuddy provides disconnected application transactions and supports escrow reservations and disconnected reservation split and transfer, implementing the protocols described in

Sec. 4 and Sec. 5 on top of the base MX disconnected object storage system [24]. To support expressive applications, following Mobisnap [20], in addition to escrow objects, MobileBuddy provides a set of additional generic leases corresponding to the locking modes supported by SQL systems. For brevity, we omit the details of the MobileBuddy system implementation in MX that are straightforward and can be found in a technical report [25, 28]. Here we describe our performance experiments.

Methodology and findings Exo-leasing provides two types of benefits for disconnected collaborators. First, the ability to obtain reservations and to validate disconnected transactions 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 application-specific costs, reduces to the general benefit of type-specific and generic locking, and has been studied before (e.g. the results in [20] apply). Second, obtaining a reservation from a nearby collaborator, instead of the server is advantageous 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 [24], and others [3, 18] apply). We do not repeat the evaluation of the known *benefits* of exo-leasing. Instead, we evaluate the *overhead* introduced by the validated DON transactions, the new feature supported by exo-leasing that has not been studied before.

We evaluate this overhead using the example mobile salesman scenario described in Sec. 5, considering two possible situations. In one case, transactions run with sufficient leases for all the objects they access and therefore validated transactions can provide a practical benefit to the client. In the other case, transactions run with insufficient leases and therefore the validations fails. Our experiments highlight the performance differences between the two cases. The findings, using transactions running 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). For transactions that benefit, the overhead is reasonable. 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 testbed 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 suffi-

cient 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 [4]; 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 checking 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 Validated transactions incur overhead 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.

We distinguish between the validated transaction overhead in the situation where client holds insufficient leases, and the overhead in the situation where client holds sufficient leases, referring to the former as *Penalty* and later as *Cost*. *Penalty* is our main concern since in this case there is no benefit to the client. For *Cost*, our concern is whether the overhead is reasonable.

Penalty Consider the 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, 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. In terms of absolute time, in the worst case 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 obtains 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%. This includes $InvalidationChecks$, adding 7% extra relative the total reconnection validation time, and $CleanupHandlers$, adding 25% extra to total validation time. In absolute time, *DurableCommit* adds 305ms to the total reconnection validation. A realistic workload is unlikely to have that many escrow leases so the overhead will be lower.

Cost In this situation, 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 overhead $Tentative$, defined the same way as in *Penalty* above. This overhead is high, 47% in our experiment. Recall, the difference between this overhead and the corresponding one in *Penalty* situation 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 1 summarizes the client-side overheads *Penalty* and *Cost* for validated DON transactions.

Table 1. Overheads of validated DON transactions

	Cost	*Penalty*
$TentativeCommit$	47 %	9%
$Transfer$	-	$TentativeCommit$ * number of transactions
DurableCommit	32%	32%

Two things to note. 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. In addition, note that client-side *DurableCommit* overhead is also incured 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 Our experiment indicates that if the client obtains escrow leases but does use them, the penalty of validated DON transactions is 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.

7 Related Work

Our work blends a number of prior ideas, optimistic disconnected client/server systems, cooperative caching, escrow synchronization and multi-level transactions. 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 are optimistic and handle conflicts after-the-fact. Coda servers [11] handle conflicting directory updates in a type-specific way. Coda clients handle conflicting updates to files using application-specific resolvers (ASR) [13], as do Ficus clients [21]. Exo-leasing differs from ASR because it avoids conflicts (in the normal case) by coordinating *in advance*, enabling disconnected validation.

MX [24] introduced disconnected cooperative caching, a feature allowing a mobile client to transfer consistent objects to another client without contacting a server. Ensemblue [18] mobile appliance system, PRACTI [3] replication framework, and Sailhan et al [22] also provide this feature. MobileBuddy is implemented on top of MX. Most peer-to-peer systems that transfer mutable objects support weak consistency. Lazy Replication [14] and Bayou [27] provide strong consistency for objects and allow to handle conflicts in a type-specific way. The mobile epidemic quorum system [10] provides multi-object transactions with standard locking.

Multi-level transactions and escrow have attracted significant research interest (most relevant approaches identified below), but no commercial systems that we know about have deployed these techniques. The need to modify the concurrency engine in the server has been the principal barrier. Weikum [30] proposed multi-level transactions with open nesting in a locking system, Lomet [15] described multi-level recovery. Unlike these systems, our base transactions are optimistic, similar to Manon et al [16], and our recovery approach handles leases [6] rather than locks. The middleware implementation [19] of the J2EE Activity Service increases concurrency for long-running connected transactions using semantic locks [7], as does the promises system [8]. A position paper [23] shows how to achieve a similar benefit for both long-running transactions and snapshot queries using exo-leasing with general type-specific synchronization [26]. Escrow synchronization was introduced by O'Neil [17] and extended to replicated systems by Kumar and Stonebraker [12]. Walborn et al [29] generalizes escrow to fragmentable and reorderable data types. The approach in Mobisnap [20] 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 all other proposals, Mobisnap implements the type-specific synchronization at the server.

8 Conclusion

This paper attacks a pracrical problem in the mobile computing space, namely, how to support escrow synchronization in a 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 split and transfer mechanism that can aid collaboration in disconnected groups and is enabled by exo-leasing. 3) It presents performance measurements of MobileBuddy, a prototype escrow synchronization and reservation transfer system based on exo-leasing, evaluatinf the client-side overhead of running disconnected validated transactions.

Acknowledgments We thank the anonymous referees for helpful suggestions and Butler Lampson, Barbara Liskov, and Mike Stonebraker for useful comments.

References

1. 'emulab.net', the Utah Network Emulation Facility. supported by NSF grant ANI-00-82493.
2. A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proc. of the ACM SIGMOD*, May 1995.
3. N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *Proc. of the NSDI*, April 2006.
4. M. Carey and et al. A Status Report on the OO7 OODBMS Benchmarking Effort. October 1994.
5. D. Gifford and J. Donahue. Coordinating Independent Atomic Actions. In *Proc. of IEEE COMPCON Digest of Papers*, February 1985.
6. C. Gray and D. Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proc. the 12th SOSP*, October 1989.
7. J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. 1993.
8. P. Greenfield, A. Fekete, J. Jang, D. Kuo, and S. Nepal. "isolation support for service-based applications: A position paper". In *Proc. of CIDR*, January 2007.
9. R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected Operation in the Thor Object-Oriented Database System. In *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.
10. J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi. Epidemic quorums for managing replicated data. In *Proc. of the IEEE ICPCC*, February 2000.

11. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM TOCS*, February 1992.
12. A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. *ACM SIGMOD Record*, 17(3):117–125, June 1988.
13. Puneet Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Workshop on Workstation Operating Systems*, pages 66–70, 1993.
14. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. In *ACM TOCS 22(3)*, November 1992.
15. D. B. Lomet. Mlr: A recovery method for multi-level systems. In *Proc. of ACM SIGMOD*, June 1992.
16. Y. Ni, V. Menon, A. Adl-Tabatabai, A. Hosking, R. Hudson, E. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proc. of the PPOP*, November 2007.
17. P. O'Neil. The escrow transaction method. *ACM Transactions Database Systems*, 11(4):406–430, June 1986.
18. D. Peek and J. Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *Proc. of OSDI*, November 2006.
19. F. Perez-Sorrosal, M. Patino-Martinez, R. Jimenez-Peris, and J. Vuckovic. Highly available long running transactions and activities for j2ee applications. In *Proc. of the IEEE ICDCS*, 2006.
20. N. Preguica, J. L. Martins, M. Cunha, and H. Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *Proc. of the 1st MobiSys*, May 2003.
21. P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the ficus file system. In *Proc. of the Usenix Technical Conference*, 1994.
22. F. Sailhan and V. Issarny. Cooperative caching in ad hoc networks. In *Proc. of the 4th Mobile Data Management Conference*, January 2003.
23. L. Shrira and S. Dong. Exosnap: a modular approach to semantic synchronization and snapshots. In *Proc. of the 2nd Workshop WDDDM, EuroSys '08, Glasgow, UK*, March 2008.
24. L. Shrira and H. Tian. MX: Mobile Object Exchange for Collaborative Applications. In *Proc. of ECOOP*, July 2003.
25. L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. Technical Report MSR-TR-2008-112, Microsoft Research Silicon Valley, August 2008.
26. L. Shrira and H. Xu. Snap: a non-disruptive snapshot system. In *Proc. of the ICDE*, Tokyo, Japan, 2005.
27. D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the ACM SOSP*, 1995.
28. Hong Tian. *MX: Mobile Object Exchange for Collaborative Applications*. PhD thesis, Brandeis University, 2005.
29. G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proc. of the SRDS*, 1995.
30. J. Weikum. A theoretical foundation of multi-level concurrency control. In *Proc. of the ACM PODS*, 1986.