Transaction Management for Mobile Objects using Optimistic Concurrency Control

Atul Adya

July 1994

This work was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136.

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Transaction Management for Mobile Objects using Optimistic Concurrency Control

Atul Adya

Abstract

We present the design of a new transaction mechanism for an object-oriented database system called Thor. We also describe a mechanism that allows objects to migrate from one server to another.

Our transaction management approach is different from other systems because we use optimistic concurrency control to provide serializability. Optimistic schemes have been suggested in the literature but they do not address issues such as space and logging overheads. In this thesis, we consider these problems and propose a scheme that has low space overhead per object and also has low delays. We take advantage of system characteristics such as loosely synchronized clocks and high availability to achieve these goals. We also present a novel mechanism that allows applications to increase the transaction throughput by overlapping the commit of a transaction with the execution of the next transaction.

Our work on object migration is different from previous work because we provide transaction semantics with respect to movement of objects; if a user moves a set of objects, our scheme guarantees that either all or none of the objects are moved to their destination sites. In addition, object migration is orthogonal to reading and writing of objects; this feature avoids unnecessary aborts caused by conflicts between the migration primitives and reads/writes. We accomplish these goals by a simple modification to the basic validation scheme and commit protocol.

Keywords: transaction, optimistic concurrency control, validation, object mobility, two-phase commit, distributed systems, object-oriented databases.

This report is a minor revision of a Master's thesis of the same time title submitted to the Department of Electrical Engineering and Computer Science on January 31, 1994, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science. This thesis was supervised by Professor Barbara H. Liskov.

Acknowledgments

I would like to thank Barbara Liskov, my thesis supervisor, for her patience, counsel and support. Her insightful comments have been most valuable for my thesis. The presentation of this thesis has been significantly improved because of her careful and critical proof-reading.

Bob Gruber helped me with his critical comments on different aspects of the thesis. He also proof-read some of the chapters and helped me make my presentation better.

My discussions with Mark Day provided me with insight into different aspects of the system. His comments also helped me in designing my schemes.

Umesh Maheshwari provided me with moral support during my work on this thesis. He also discussed various design issues with me and pointed out several problems during the early part of this thesis.

My officemate Phil Bogle proof-read some of my chapters. He and my other officemate, Quinton Zondervan, have been a great source of encouragement for me. My numerous technical and non-technical discussions with them have been most useful and enjoyable.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents the design of a new transaction mechanism for a distributed client-server system. Most previous systems have used pessimistic concurrency control for transaction management. Our approach is different from these systems since we use optimistic concurrency control to provide serializability. In addition, the thesis presents the design of a mechanism that allows objects to move from one node to another. Our object migration approach is integrated with the concurrency control mechanism; this strategy allows us to provide transaction semantics with respect to movement of objects.

Our work has been done in the context of a new object-oriented database system, Thor. Thor [Liskov93] is a distributed system based on the client-server model. It provides persistent and highly available storage for objects by storing them at the servers. Each client runs on a workstation and executes its operations as part of an atomic transaction. Clients can access objects from multiple servers and operate on them. Objects are cached at clients in order to improve system performance. Details of the Thor system architecture are discussed in Chapter 2.

The remainder of this chapter elaborates on our contributions and discusses related work. Section 1.1 presents the motivation and an overview of our transaction management strategy. Section 1.2 discusses the issue of migrating objects among servers. Finally, Section 1.3 provides an overview of the remainder of this thesis.

1.1 Transactions and Concurrency Control

Transactions [Gray93] are a convenient mechanism for building reliable distributed systems in the presence of concurrent access and failures. They allow operations on objects to be grouped together and provide the atomicity guarantee, *i.e.,* either all or none of these operations are performed on the database state.

Any system that supports transactions needs a concurrency control mechanism to coordinate them. Schemes that have been suggested to achieve the effect can be broadly classified into two categories — *pessimistic* concurrency control and *optimistic* concurrency control schemes. A pessimistic scheme is based on the notion that any access to an object must be preceded by a request asking for permission to retrieve the object in the desired mode, *e.g.,* read or write. A strategy based on optimistic concurrency control *optimistically* assumes that there will be few or no conflicting actions and allows immediate access to the object. Work done as part of a transaction is synchronized or *validated* at the end of the transaction. A transaction validates successfully and is committed if it can be serialized with other transactions in the system. If the transaction is not serializable, it is aborted. Herlihy [Herlihy90] has classified pessimistic and optimistic schemes respectively as *asking permission first* and *apologizing later*. Either of the two schemes can be employed for centralized or distributed systems.

Section 1.1.1 presents the client-server model and the concurrency control schemes used by past and existing systems. Section 1.1.2 discusses the protocols suggested in the literature for committing transactions. Finally, Section 1.1.3 describes our approach to transaction management.

1.1.1 Concurrency Control Schemes for Client-Server Systems

Most of the current distributed database systems are implemented using a client-server architecture. Objects are stored persistently at one or more server machines. Clients fetch objects from server machines, operate on them locally and send back any modifications to the server. Such architectures improve system performance by utilizing the processing power of client machines; the server's load is reduced by performing as much computation as possible on client machines. To reduce fetch delays, these systems cache objects at the client. Prefetching is another technique that is used to improve the system performance. Prefetching refers to the idea of fetching objects from a server even before they are required. When a client sends a fetch request to a server, the latter returns the desired object and a few extra objects in anticipation that the client will use them soon. Thus, when the client actually needs an object, it is already present in the local cache. A concurrency control algorithm designed for a client-server based system must take caching and prefetching into account; it should not nullify the advantages of these strategies.

A variety of optimistic and pessimistic schemes have been developed for providing concurrency control in client-server systems. However, systems in the past have used pessimistic schemes for transaction management. In earlier client-server architectures, due to small client caches, little or no caching of data was done across transactions; if the same piece of data was accessed by subsequent transactions at a client, it had to be fetched again from the relevant server. But due to recent increases in memory sizes, newer databases try to improve the system performance by caching data across transactions; existing machines can cache a significant fraction of a client's working set. A simple pessimistic scheme in which locks are not cached across transactions loses the advantage of client caching; a client has to send a lock request to the server even if it has the object in its cache (we will refer to this scheme as *regular locking*). To improve system throughput, pessimistic schemes such as no-wait locking, callback locking and optimistic locking have been suggested [Franklin92]. These locking techniques make "optimistic" assumptions about lock acquisition and lock sharing to reduce synchronization overheads. Optimistic schemes take advantage of client caching by not requiring any message to be sent to the server if the object is already present in the client cache.

Callback locking takes advantage of client caches and retains read locks after committing a transaction. When a server receives a client request to acquire a write lock, it asks the relevant clients to release read locks. In *no-wait locking*, a client requests a lock but does not wait for the reply from the server; client processing proceeds in parallel with the lock request. If the lock request fails, the client transaction is aborted. These schemes have been used by some systems, *e.g.,* Symbolic's Statice system [Weinreb88] uses no-wait locks and ObjectStore [Lamb91] uses callback locking. In *optimistic locking*, a client does not acquire any locks before any operation; at the end of the transaction, a server waits for clients to release conflicting locks on relevant objects and commits the transaction. Note that this scheme is different from a classical optimistic scheme where instead of acquiring locks at the end of the transaction, the servers execute a validation algorithm to check if the transaction has violated any serializability constraints.

Franklin [Franklin92] and Wang [Wang91] have conducted studies to compare the performance of these schemes for different workloads in a client-server model. These concurrency studies indicate that callback, no-wait and optimistic locking provide higher throughput than regular locking for most workloads. Furthermore, optimistic locking performs as well or better than the other approaches for most workloads. This is so because it exploits client caching well and also has relatively lower network bandwidth requirements. No-wait locking avoids lock synchronization delays but it still needs to send lock request messages to the servers; an optimistic scheme avoids these messages also. Due to lower message requirements, optimistic locking performs better or as well as callback locking for most cases. In the latter scheme, lock recall messages are sent during the transaction's execution whereas the former scheme groups these messages and sends them at the end of the transaction. Note that these locking-based schemes incur some processing and message overhead due to global deadlock avoidance or detection. On the other hand, a validation-based optimistic scheme does not suffer from this problem. In such a scheme, deadlocks cannot occur; one of the transactions in the wait-cycle is automatically aborted at validation time.

Franklin's studies show that optimistic locking performs poorly in high contention environments. In pessimistic schemes, transactions that access "hot spot" objects (objects that are frequently modified and shared among many clients) are delayed by the transaction manager till the desired lock can be granted. Optimistic schemes allow such transactions to proceed; conflicts are detected later causing most of these transactions to abort. The poor performance of optimistic locking for hot spot objects results from the fact that these schemes convert waiting on locks to transaction abort. For high-contention environments, locking is desirable but for other workloads, optimistic schemes have a better performance. Therefore, to support both low and high contention workloads, an adaptive scheme can be designed that usually uses optimistic concurrency control but dynamically changes to locking for hot spot objects. Such a technique is being developed by Gruber [Gruber94].

Optimistic schemes have been discussed in the literature but we do not know of any multi-server distributed system that serializes transactions using such an approach. The seminal paper on optimistic concurrency control by Kung and Robinson [Kung81] motivates the need for optimistic schemes and presents the idea of validation. Their centralized server scheme for serial and parallel validation is generalized to a distributed system in [Ceri82]. Häerder [Häerder84] has developed the notion of forward and backward validation. Optimistic schemes have also been extended to exploit the semantics of abstract data types [Herlihy90]. Gruber [Gruber89] has suggested validation-based optimistic schemes for the nested transaction model. A system implementation that caches objects and uses a classic optimistic scheme is Servio Logic's Gemstone [Maier86]. Gemstone is a distributed system that allows multiple clients but objects can be stored at only one server. The Jasmin database machine [Fishman84] also uses optimistic concurrency control for serializing transactions; it too is a centralized server system. Their concurrency control control algorithm was later extended to the distributed case [Lai84]. However, they use multiple versions and also serialize transactions at a site in the order they are received.

Some systems use a combination of optimistic and pessimistic schemes for serializing transactions. A hybrid optimistic-pessimistic scheme has been suggested in [Lausen82]. Another hybrid scheme has been proposed in [Rahm90]; this scheme uses optimistic concurrency control for a transaction but switches to locking if the transaction aborts. The object-oriented database Mneme [Moss90] provides support for such schemes. The idea of hybrid concurrency control schemes has been applied to file systems also. The Amoeba file system [Mull85] employs such a technique for modifying files; updates on a single file are serialized using optimistic concurrency control whereas locking is used to modify multiple files.

A multiversion pessimistic scheme that does not require read-only transactions to synchronize with other transactions was suggested in [Weihl87]. An optimistic strategy that achieves the same effect has been presented in [Agarwal87]. Maintaining multiple versions may permit more transactions to commit than a single version scheme. However, it complicates transaction processing and reduces the effective cache size at the server/client; multiple versions of objects have to maintained consuming more storage in the client cache.

The nested transaction model has been explored and discussed in the literature. The idea of nested transactions was proposed in [Moss81]. Reed [Reed83] describes a timestamping strategy for serializing transactions in this model. Systems such as Argus [Liskov84] and Camelot [Spector87] provide a computational model that supports nested transactions using pessimistic locking. An optimistic scheme for this model has been presented in [Gruber89]. To simplify the transaction model, Thor does not support nested transactions.

1.1.2 The Commit Process

When a client commits a transaction, the system has to ensure that all servers agree on committing or aborting the transaction. This effect is usually achieved with the help of a 2-phase commit protocol [Gray79, Mohan83, Lindsay84]. Many variations have been suggested to optimize this protocol, *e.g.,* presumed-abort/commit strategies. Mohan [Mohan83] has adapted the 2-phase commit protocol for a tree of processes. The coordinator log protocol suggested in [Stamos89] does not require a participant to flush the prepare record. Group commit [DeWitt84] strategies have been suggested in which log flushes of multiple transactions are grouped together to alleviate disk bandwidth problems; the Cedar system [Hagmann87] uses this strategy. Non-blocking and 3-phase commit protocols have also been studied [Skeen81]. Camelot has implemented a non-blocking protocol but its performance is much worse than a 2-phase protocol [Duchamp89]. Non-blocking protocols are not of practical interest and systems rarely implement them.

During the commit protocol for pessimistic systems, read-locks can be released during the first phase but write-locks have to be retained till the second phases completes. Levy [Levy91] has developed a protocol where all locks can be released after preparing a transaction. But this protocol provides a weaker guarantee than serializability, making the transaction model quite complicated.

Lomet has suggested a timestamping scheme [Lomet93] for the two-phase commit protocol in a pessimistic system where a transaction commits at some nodes and continues processing at other nodes. In this approach, the participants vote a timestamp range to the coordinator for the transaction's commit time. The latter chooses a value and sends it as part of the commit decision. Some of the complexity in Lomet's protocol arises from the fact that the transaction timestamp is being chosen in the protocol's second phase rather than the first phase. Furthermore, allowing read locks to be released at the end of phase one adds more complexity to the algorithm.

Lampson and Lomet [Lampson93] have suggested a presumed-commit strategy that reduces the number of background log writes and messages for the normal case. But these gains are achieved at the cost of retaining some commit information forever. Furthermore, a presumed-commit strategy cannot be used in a system where along with the coordinator's decision, some other information has to be sent as part of the phase 2 message. For example, in Thor, the coordinator needs to send information about newly created objects to the participants along with its commit decision.

1.1.3 Our Design for Transaction Management

As stated earlier, the studies conducted by Franklin and Wang have shown that an optimistic scheme performs better than pessimistic schemes in environments where there is low contention on objects. We have made this assumption about the workload and designed an optimistic concurrency control scheme for a client-server distributed system. Our design is based on validation and not locking. However, we expect a validation-based optimistic scheme and optimistic locking to have similar performance because both schemes verify serializability constraints at the end of a transaction. Our optimistic scheme may have lower message and bandwidth requirements than optimistic locking since our scheme does not require messages to be sent to clients at commit time.

In this thesis, we adapt the optimistic schemes that have been suggested in the past and propose a validation strategy that truncates transaction history information frequently without causing unnecessary aborts. We assume the availability of an external service such as NTP [Mills88] that provides loosely synchronized clocks. This assumption allows a server to generate an appropriate timestamp for a committing transaction; it has also helped us in simplifying the validation process. In addition, loosely synchronized clocks help in reducing the logging requirements for read-only transactions and the space overhead for each object.

All techniques developed in this thesis aim to reduce application observable delays (also called as foreground delays) as much as possible. In Thor, an application waits for the transaction result until phase one of the 2-phase commit protocol has been completed. Thus, it is important that the time taken by this phase is minimized. This goal is achieved by optimizing the validation algorithm and by decreasing foreground network delays. Like other optimistic schemes in the literature, our scheme also allows multiple transaction validations to proceed simultaneously at a server; this optimization reduces synchronization delays at a server. Foreground network delays are reduced by optimizations such as early send, Stamos's coordinator log protocol and short-circuited prepare.

With relatively few changes to our protocol, we can support the scenario described by Lomet (discussed in Section 1.1.2). There are no complications regarding the release of read locks in an optimistic scheme. Furthermore, assigning timestamps at the beginning of phase one using loosely synchronized clocks also helps us in avoiding most of the complexity of Lomet's protocol.

A novel contribution of this thesis is the idea of *asynchronous commit.* The asynchronous commit facility gives more flexibility and better control to applications; an application can overlap a transaction's commit process with the next transaction's normal processing to reduce application observable delays. In the usual case, a client sends a commit message to the servers and waits for the result. In the asynchronous commit case, the commit call returns immediately; this allows the application to proceed with the next transaction. As a result, the commit time delay is the same as the delay observed for any normal operation since the commit protocol is executed in the background. When the asynchronous commit call returns, the application does not know the result of the transaction commit; it needs some way to inquire about the commit result later. Thus, the application interface has to be enhanced to support this strategy. We suggest possible extensions to the application interface and demonstrate how the code-structure may be altered to use asynchronous commit.

1.2 Object Migration

Any system that is intended to be used for a long time such that it outgrows its initial configuration must provide a way of migrating objects. The initial object placement by an application or the system may not be suitable after some time. Thus, to improve performance, applications need a way for reconfiguring the object placement. We discuss a strategy that allows applications to migrate objects from one server to another. Our object migration scheme is integrated with the concurrency control mechanism; we provide strong semantics of atomicity and serializability with respect to object migration.

Object migration can be used by an application to cluster objects at one or few servers. Clustering objects at one server reduces the number of inter-server or external references. This leads to more effective prefetching because a significant fraction of an application's working set is brought to the client machine from that server as prefetched objects instead of a client sending explicit messages and waiting for them (assuming that the prefetching strategy prefetches objects referenced at the same server by a fetched object). Fewer external references are beneficial for the distributed garbage collection algorithm also [Mah93]. Inter-server references can be reduced by moving an object to a server where most references to it reside or vice-versa. Apart from reducing inter-server references, clustering an application's objects at a single server has the advantage that a 2-phase commit protocol is not needed to commit a transaction; this decreases the commit time delay by a log flush and a network roundtrip delay.

An application may migrate objects due to the physical movement of the corresponding (physical) entities; objects are moved to a server that is physically near the client site. Object mobility may also be used to balance the object load across various servers; if a server becomes loaded with a large number of objects, its load can be reduced by moving some of its objects to lightly loaded servers. Chapter 5 discusses other applications for which migration may be a useful facility.

A semantic issue that arises concerning object migration is whether locating and moving objects are related to reads/writes or not. We argue that making object migration primitives independent of reads/writes is not only more intuitive but also avoids a certain class of spurious aborts; we present a design that supports these semantics. Another interesting semantic issue is whether object migration is part of the transaction mechanism or not. We believe that atomicity and serializability with respect to object migration offers elegant semantics and makes it easier for users to reason about their programs. To provide serializability with respect to migration, we adapt our validation scheme for read/writes and use it for object locates and moves. The 2-phase commit protocol is also modified to guarantee atomic migration of objects.

Our work on object migration is different from earlier work since our strategy is integrated with the concurrency control mechanism; we provide transaction semantics with respect to object migration also. The Hermes [Black90] and Emerald [Jul88] systems support object mobility but they do not have a client-server model like Thor and they do not support transactions. The design suggested for a pessimistic system in [Tam90] supports transactions in a model where objects are migrated from one site to another if the latter site wants to modify the object. This model is neither a client-server model nor is it possible to move a set of objects to a specific site. Research has been conducted in the area of process migration for various operating environments. Systems such as Sprite [Douglis91], V [Theimer85], DEMOS/MP [Powell83] and Accent [Zayas87] support this facility. Some of the problems faced in process migration schemes are similar to the ones faced in Thor, *e.g.,* forwarding pointers have to be left at the old site and all relevant operations have to routed to the new site. But there are other issues that are pertinent only to our approach where object migration has been integrated with the concurrency control mechanism.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 describes the system architecture of Thor and introduces the relevant terminology for later chapters. We also describe an application's view of Thor and how application programs interact with the Thor universe.

In Chapter 3, we present the commit process and our basic validation scheme. We suggest ways of reducing the concurrency control space overhead for each object. A technique to avoid a foreground log flush at a read-only participant is also described. We use loosely-synchronized clocks to achieve these optimizations; these clocks help us in simplifying the validation process also.

Various optimizations to reduce application observable delays at commit time are discussed in Chapter 4. We develop a scheme that allows the transaction manager at a server to validate multiple transactions and install object modifications concurrently. A technique to avoid a foreground log flush at each participant is also discussed; this strategy is based on Stamos's coordinator log protocol. We explore the semantics and implementation of asynchronous commit in this chapter. The application interface has to be modified to support this facility. We discuss these interface changes and illustrate how the code structure may be altered to take advantage of asynchronous commit.

Chapter 5 presents our design for object migration. The primitives for locating/moving objects and their semantics are discussed in this chapter. We motivate the need for atomicity and serializability with respect to object migration and show how object mobility can be integrated with the concurrency control mechanism in a client-server system. We also discuss the changes that are made to the 2-phase commit protocol for implementing object migration.

Finally, Chapter 6 concludes the thesis and mentions the areas for future work.

Chapter 2

System Architecture of Thor

This chapter gives an overview of the Thor object-oriented database system. We discuss only those aspects of Thor that are relevant to this thesis; for a complete description, see [Liskov93]. The terminology and conventions developed in this chapter are used in the remainder of the thesis.

Thor is a new object-oriented database management system (OODBMS) that can be used in heterogeneous distributed systems and allows programs written in different programming languages to share objects. Thor provides users with a universe of persistent objects, *i.e.,* objects survive in spite of failures. These objects are also highly available; they are likely to be accessible when needed. Thor provides a persistent root for the object universe. An object becomes persistent if it becomes accessible from the persistent root. If an object becomes unreachable from the root, its storage is reclaimed by a distributed garbage collector [Mah93]. Each object has a globally unique id called *oid*. Objects contain data and references to other Thor objects. They are encapsulated so that an application using Thor can access their state only by calling their methods. In this thesis, we assume that the set of methods available to users are just read and write. Thor supports transactions that allow users to group operations so that the database state is consistent in spite of concurrency and failures.

Objects are stored at server nodes that are different from the nodes where application programs run. The Thor system runs on both the application site and the server site. The component that runs on the server side manages the storage of persistent objects. For each application, there is a dedicated process called *frontend* that handles all the application requests. A frontend machine is assumed to have no persistent storage. The universe of Thor objects is spread across multiple servers and application programs can access these objects by interacting with their frontends. Figure 2-1 gives a schematic view of Thor. Although objects are stored at multiple servers, an application is given the view that the Thor object universe is a single entity.

Thor is intended to be scalable, *i.e.,* a large number of servers or frontends may exist in the Thor universe at any given time. Furthermore, the object database and the application sites may be separated by a wide-area network. The Thor design also takes account of the fact that objects may persist for a long time (*e.g.,* years). As the system is used, applications may want to change the system configuration. The object migration facility helps an application perform the desired reconfiguration.

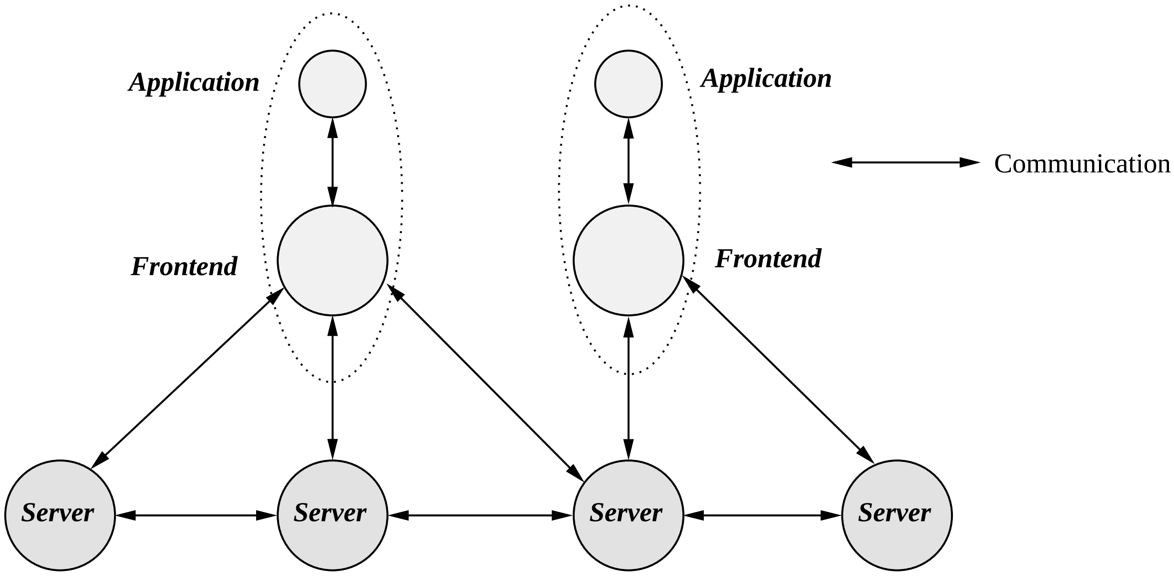


Figure 2-1: Applications, frontends and object repositories in Thor

2.1 Object Servers

Each server manages some subset of the Thor object universe. A server stores a *root directory* object that contains references to other objects or other directories. Applications can ask for a server's root directory and access objects from that server. But it not necessary to access objects through the root directory only; objects can be accessed by queries or simply following pointers between objects.

Objects stored at a server may contain references to objects at the same server or at other servers. A reference, also called *xref*, is implemented using a server's id (each server has a globally-unique id called *server-id*) and a local name within that server called *oref*. The tradeoffs associated with location-independent and location-dependent references and the reasons for choosing the latter scheme are discussed in [Day94]. Given a reference, the corresponding object can be accessed by sending a request to the relevant server. This server uses the oref value to fetch the requested object.

For garbage collection purposes, each server keeps track of information about the objects that have been cached at different frontends in a table called the *frontend-table*. A server A also maintains a table called the *inlist* that keeps track of objects at other servers that have references to objects at A.

Each server is replicated to make objects highly available. We plan to use a primary copy scheme for replication [Oki88]. In this scheme, each server's objects will be replicated at a number of machines. For each object, one of the machines that has a copy of that object will act as the *primary* server and the others act as *backup* servers. The frontend always interacts with the current primary server. All servers have uninterruptible power supplies (UPS's) in addition to disks; the UPS's protect against power failures, which are the most likely cause of a simultaneous server crash. The UPS's allow us to view data as safely recorded on stable storage as soon as it resides in volatile memory at the primary and backups; the data is written to the disk at some later time. Thus, writing to stable storage (*e.g.,* flushing a log record) is equivalent to a network roundtrip delay. The primary and backups exchange messages periodically. These messages, referred to as *liveness* messages, are used by the replication scheme to determine whether either of the primary or backup process has failed. In the rest of the thesis, unless indicated otherwise, we use the term server to refer to the primary server. We also assume that there is one backup server for every primary.

2.2 Applications and Frontends

The system creates a frontend process for an application whenever the latter wants to access objects from the Thor universe. When the frontend interacts with a server for the first time, it creates a *session* with that server. The frontend and the server then maintain information about each other until the session terminates. The frontend fetches objects by sending the xrefs of the desired objects to the relevant servers. To start accessing its persistent objects, the application can access the root directory of a server without knowing its xref; it can then navigate through the database.

A frontend process is usually created on the same machine as the application. An application program never obtains direct pointers to objects; instead, a frontend issues *handles* that can be used to identify objects in subsequent calls addressed to it. These handles are meaningful only during an application's particular session with its frontend. An application program may be written in any programming language. If the language is type-safe, the application and the frontend may run in the same address space. But programs written in unsafe programming languages may corrupt the frontend's data; such an application must run in a separate address space and interact with its frontend by means of messages.

The frontend is responsible for executing application calls. It *caches* copies of persistent objects to speed up the application. An object copy stored in a frontend cache is called a *shadow version* and its stable copy at the relevant server is referred to as its *base version*. Objects in the frontend cache may contain references to objects that are not in its cache. When an application makes a call in which an attempt is made to follow such a reference, a fetch request is sent to the relevant server. The server sends the requested object to the frontend along with some additional objects that might be required by the frontend in the near future. This technique is referred to as *prefetching*. The frontend may send some prefetching hints to the server to help the latter in selecting the extra objects.

One possible prefetching strategy is to prefetch objects by following the references of the fetched object. Figure 2-2 shows a scenario in which object x refers to objects y and

z. Shadow versions of x and z exist in the frontend cache but y is not present. When the reference to y is accessed, an "object fault" occurs and y is brought in the frontend cache. Along with y's fetch, object u is prefetched by the frontend.

Any operation for which the application waits is said to have been executed in the *foreground* whereas operations done in parallel with the application's computation are said to have run in the *background*. For example, a fetch request occurs in the foreground since the application waits until the object has been brought into the frontend cache. On the other hand, the commit protocol's second phase occurs in the background since the application continues its processing in parallel with it.

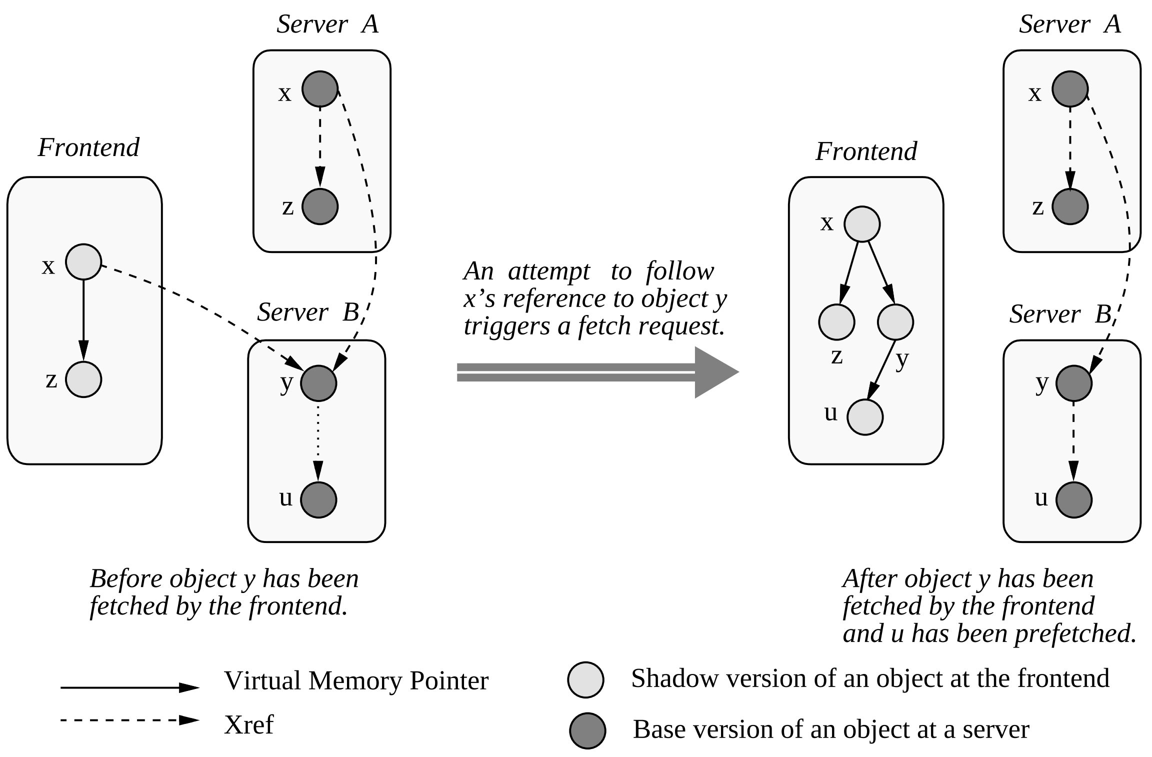


Figure 2-2: Accessing an object not present in the frontend cache

An object fetched from a server contains references in the form of xrefs. The frontend converts these references into virtual memory pointers of cached objects for better performance. This process is termed as *swizzling* [Moss90]. The frontend may swizzle an object x on receiving it from the server or it may swizzle x lazily, *i.e.,* swizzle the references when x is accessed for the first time by the application. Furthermore, it may or may not swizzle a complete object. Different options for swizzling and their associated performance tradeoffs are discussed in [Day94].

2.3 Transactions

All operations of an application are executed as part of a Thor transaction; a new transaction starts automatically after the current transaction commits or aborts. While performing its computation, an application may read or modify objects in its frontend's cache. Objects in the frontend cache for which a stable copy exists at some server are said to be *persistent*; other objects are referred to as *non-persistent* objects. When an application reads or writes a persistent object, its frontend records this operation. When the application commits the transaction, its frontend sends copies of modified objects to the relevant servers. These modified values are installed if the transaction succeeds in committing. The frontend determines the non-persistent objects that are reachable from the set of modified objects. It sends these objects to the servers along with its preference of where each of these objects should reside. These non-persistent objects become persistent if the transaction commits and are called *newly-persistent* objects.

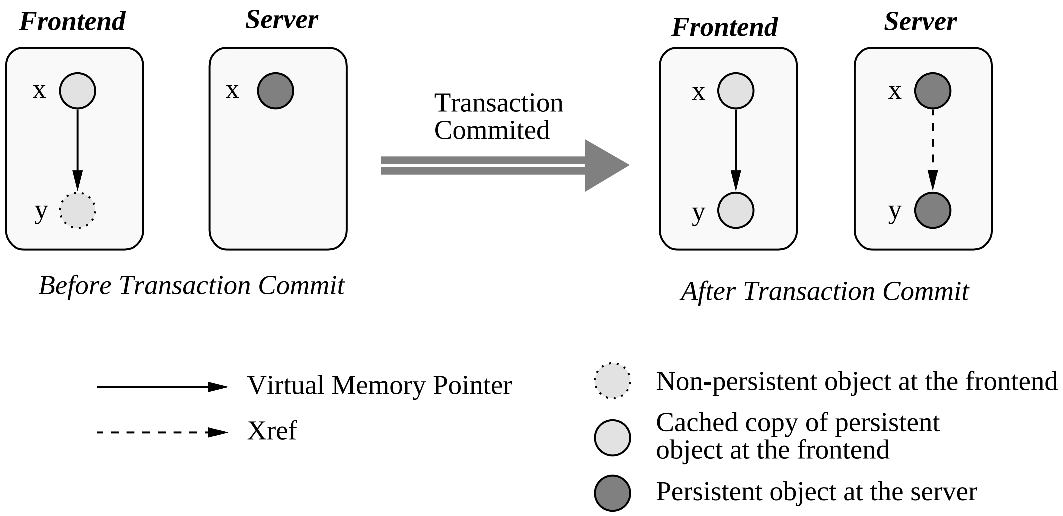


Figure 2-3: Creation of a newly-persistent object

To commit a transaction, a frontend sends the transaction information to one of the servers and waits for the decision. This server, also known as the *coordinator* of the transaction, executes a 2-phase commit protocol and informs the frontend of the result (details of this protocol are presented in Chapter 3). The application waits only until phase one of this protocol is over; the second phase is carried out in the background. During the first phase, a server allocates space for the newly persistent objects that will reside at that server. It also assigns oids/xrefs to these objects and sends this information to the coordinator. If the transaction commits, the coordinator informs the frontend about the new xrefs and oids. If the transaction aborts, this space, xrefs and oids are freed up for reuse. Figure 2-3 shows a scenario in which object x has been modified by a current transaction to include a reference to non-persistent object y. When the transaction commits, object y is installed at the server and the frontend also records y as a persistent object.

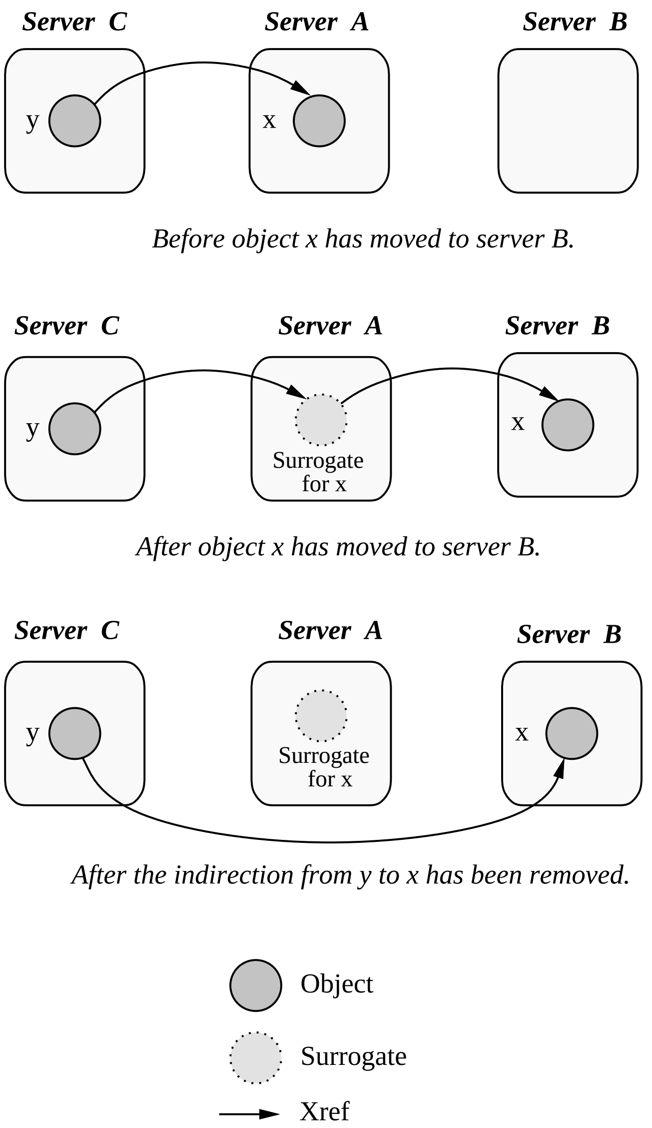


Figure 2-4: Moving an object from one server to another

Before sending objects to a server, the frontend has to convert the virtual memory pointers contained in the modified and newly-persistent objects to xrefs. This process is referred to as *unswizzling*. References to newly-persistent objects cannot be unswizzled to xrefs since these objects are assigned xrefs during the commit process. Unswizzling of pointers to newly-persistent objects has to be done by the servers that contain these references. To make this work possible, the coordinator sends the xref information about the newly-persistent objects to the relevant servers as part of its second phase message.

After a server has installed a transaction's object modifications, it sends *invalidation* messages to frontends that have cached these objects; information about which frontend has cached the relevant objects can be obtained from the frontend-table. The server can send the modified value to the frontends or simply ask them to invalidate the relevant objects. Or it may decide to send the update values of some objects and invalidate the rest. The advantages and disadvantages of these schemes for different workloads are discussed in [Franklin92]. On receiving an invalidation message for object x, a frontend aborts the current transaction if its application has read or written x; otherwise, depending on the strategy used it updates or invalidates x's cached copy. The frontend invalidates x's copy by converting it to a *frontend-surrogate*. This surrogate is a small object that just contains x's xref. The frontend-surrogate is needed because other objects at the client may refer to x's invalidated copy.

2.4 Object Mobility

Objects in the Thor universe are allowed to migrate from one server to another. When an object moves to another server it leaves a *surrogate* at the old server. This surrogate contains a forwarding pointer for the object's location at the new server, *i.e.,* the new xref of the object. If a frontend tries to fetch the object from the old server, the latter returns the surrogate to the frontend who can now fetch the object from the new server. Since an object may move again to a different server, it is possible that an object fetch might require following a chain of surrogates. These chains are snapped by the distributed garbage collector or the migration mechanism itself. The chain snapping process will be able to keep the surrogate chains short because we assume that objects move rarely and the information about object movement is propagated quickly to the relevant frontends and servers.

Figure 2-4 shows the movement of object x from server A to server B; it leaves a surrogate at its old server. Object y stored at server C contains a reference to x. Because of an indirection through the surrogate, this reference remains valid even after x has moved to B. This indirection is later snapped and then y points to x directly. The surrogate will be garbage collected when no reference to it exists in the system.

Chapter 3

Basic Concurrency Control Scheme

This chapter discusses our transaction commit strategy. It focuses on the basic optimistic concurrency control mechanism employed to validate transactions. Section 3.1 presents an overview of Thor transactions and the commit process. Section 3.2 describes the validation scheme using a transaction history. In Section 3.3, we suggest a way of truncating this history. Further optimizations to reduce the space and logging overheads are explored in Section 3.4. Finally, Section 3.5 presents our basic validation algorithm.

This chapter assumes that validation of transactions and installation of objects is done in a single critical section. This condition is relaxed in the next chapter.

3.1 Overview

3.1.1 Characteristics

Each server has a transaction manager (TM) that handles all the commit related activity. The TM ensures that transactions are committed only if they do not violate serializability. The design of the TM is based on the following characteristics of Thor transactions:

- Concurrency control is performed at the object granularity level. Any modification to an object generates a new value for it.

- Blind writes are not allowed, *i.e.,* if a transaction has modified an object x, the TM assumes that it has read x too. Blind writes are rare or non-existent in practice and making this assumption simplifies the validation algorithm.

- Two transactions *conflict* on an object if one has read the object and the other is modifying it, *i.e.,* the usual notion of read-write conflict. Note that a write-write conflict will also cause a read-write conflict since blind writes are not allowed.

System Characteristics

Our design for transaction management has been influenced by the characteristics of the operating environment. The TM optimizes the normal case processing and avoids time consuming operations on the critical path of the commit protocol. It makes the following assumptions about the operating environment:

- Network partitions and frontend/server crashes are rare. Furthermore, since each server is replicated for high availability, the likelihood of two servers not being able to communicate with each other is very low.

- Although multiple applications can execute transactions and access different servers concurrently, it is unlikely that they generate a commit request at the same instant of time. The validation algorithm has been designed to take advantage of this fact.

- We make an assumption about the workloads that there are few conflicts on objects. As a result, transaction aborts are unlikely to occur.

- Loosely synchronized clocks are available for generating timestamps. As discussed later, this feature prevents a certain class of aborts and simplifies the validation algorithm.

3.1.2 Thor Transactions

As stated earlier, all read or write operations executed by an application are run as part of a transaction. To commit its currently executing transaction T, the application invokes a commit request at its frontend and waits for the reply. The frontend initiates a 2-phase commit protocol among the *affected servers*, *i.e.,* servers whose objects have been read/written/created as part of T's execution. It receives the decision from these servers and informs the application of the outcome. If the commit succeeds, the servers guarantee that T's changes become persistent.

The above process constitutes the lifetime of a Thor transaction T and can be divided into 3 phases:

Execution phase — During this phase, T reads data items, performs computations and writes new values of objects. All modifications of persistent objects are made to a copy in the frontend's local cache. The information about objects read/modified by T is maintained by the frontend for use in the later phases.

Validation phase — This phase begins when the application asks its frontend to commit T. The frontend initiates a 2-phase commit protocol to decide whether T can be committed. The first phase of the commit protocol constitutes the validation phase. During this phase, each affected server executes a validation check (see Section 3.2) to determine if T is violating any serializability constraints.

Update phase — If T passes the validation check, its updates are made available for later transactions, *i.e.,* the new values of modified objects are installed at the relevant servers. These installations form the update phase of T.

Delays in the first two phases are visible to an application, *i.e.,* it waits while the read/write operations are being processed or the transaction is being validated. Thus, the transaction mechanism must ensure that such foreground delays are minimized. Execution phase delays can be reduced by techniques such as caching and prefetching [Day94]. Our work is mainly focused on the validation and update phases, but we ensure that execution phase operations such as object (pre)fetch are not penalized.

3.1.3 Distributed Commit Process

This section gives an overview of the two-phase protocol executed to commit a transaction. In the following discussion, the server to which the frontend sends the transaction information is termed the *coordinator*. All the affected servers, including the coordinator, are also referred to as *participants*. An affected server where no object has been created or modified as part of the transaction is called a *read-only* participant and the transaction is referred to as a *read-only-at-site* transaction for that participant. Similarly, a non-read-only-at-site transaction at a participant is also termed as an *update-at-site* transaction.

When an application commits a transaction T, the frontend sends the following information to the coordinator server:

1. **Read Object Set or ROS** — Set of objects read by T.
2. **Modified Object Set or MOS** — Set of objects modified by T. Since blind writes are not allowed, the MOS is always a subset of the ROS.
3. **New Object Set or NOS** — Set of objects that are being made persistent for the first time. The NOS objects are installed iff T commits.

The coordinator assigns a globally unique timestamp[1] to T and initiates the 2-phase commit protocol. In the first phase of the 2-phase commit protocol, the coordinator sends prepare messages to all participants. Each participant runs a serializability check and sends its vote to the coordinator. If the coordinator receives a *yes* vote from all participants, it decides to commit the transaction; otherwise, it aborts the transaction. It informs the frontend about the decision and the latter conveys the transaction's commit result to the application. As part of the second phase, the coordinator informs the participants about the transaction's commit result. Each participant logs the coordinator's decision and sends an acknowledgement to the coordinator. As an optimization, each participant sends *invalidation* messages to frontends that have cached objects modified by this transaction; these messages ask frontends to flush old copies of the modified objects from their cache. If the currently executing

[1] The need for timestamps is discussed in Section 3.2.1.

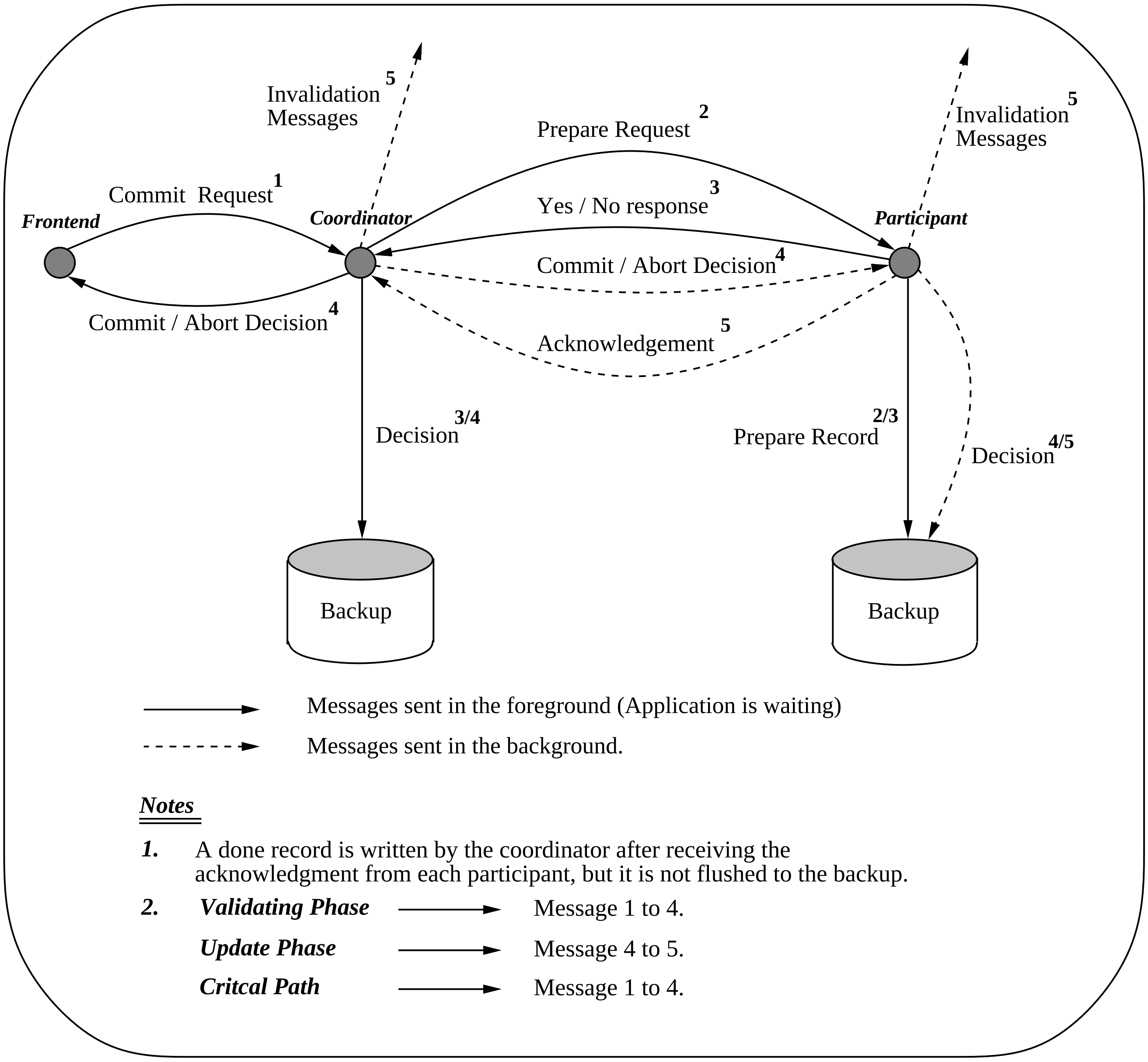


Figure 3-1: The two phase commit protocol in Thor

transaction at a frontend has read any of the objects, it is aborted. Thus, invalidation messages prevent transactions from doing wasted work.

To make the protocol resilient to crashes, each participant must log a prepare record on stable storage before sending its vote to the coordinator. It must also log a commit record on stable storage before sending its acknowledgement message to the coordinator. Similarly, the coordinator needs to log a commit record before informing the application about the commit/abort decision. Our design uses the *presumed-abort strategy* [Mohan86] and also does not require phase 2 messages to be sent to read-only participants (for details of the 2-phase commit protocol, see [Mohan86]).

Note that the frontend waits only while the first phase of the protocol is being executed. Thus, this phase of the commit protocol is said to have executed in the *foreground* (see Chapter 2). The second phase proceeds in the *background*, *i.e.,* the application does not wait for this phase to be completed.

The messages and log forces involved in committing a transaction are shown in Figure 3-1. Numbers indicate the order of messages, *i.e.,* message^{i} precedes message^{i+1}. Messages with the same numbers can be sent in parallel. A force to the backup has a superscript$^{i/j}$ indicating that it is done after receiving message^{i} but before sending message^{j}.

During the commit process, the participants also exchange information about newly created objects. As explained in the previous chapter, each participant assigns new xrefs/uids and allocates space for the objects created at its site. This allocation is done along with the validation process; if a participant is unable to allocate space for these objects, the transaction is aborted. Otherwise, if the transaction passes validation, the participant sends the new object information to the coordinator along with its vote. The coordinator merges this information from all the participants and sends it as part of the second phase. Each participant needs this information to resolve references to the newly created objects. A newly-persistent object x is installed in the second phase of the protocol. However, the frontend is informed about x's new xref after the first phase itself. Therefore, it may happen that the frontend sends a fetch request for object x before x has been installed at the server. The server can declare this fetch as invalid or delay the fetch till x has been installed.

The state of an object x stored at a server is termed the *base version* of x. This object can be cached at different frontends; each copy of x is called a *shadow version* of x. When a transaction that has modified x at frontend F commits, x's shadow version at F becomes the new base version at x's server. If a prepared transaction T is modifying x, the value of x that T is trying to install is called the *potential* version of x. Thus, we can view x having multiple versions with the base version being the latest installed version of x. We refer to the i^{th} version of x as x_i and x's base version is denoted by x_{base}.

3.2 Validation

Optimistic concurrency control schemes are designed to get rid of the locking overhead of pessimistic schemes. Instead of performing checks during the read/write operations, these schemes defer the burden of concurrency control till the validation phase. In the validation phase, the TM at each participant verifies that the transaction has not violated any serializability constraints; this process is termed *validation* or *certification*.

There are two kinds of validation — *forward validation* and *backward validation*. The former validates an incoming transaction T against all the concurrently executing transactions and ensures that none of them is invalidated by T. Backward validation involves validating T against prepared or committed transactions; T fails validation if any of these transactions has invalidated T's operations.

Forward validation schemes have advantages over backward validation schemes:

- An active transaction never reads a stale value; applications never see an inconsistent state of the database. If backward validation is being used, the programmer must be aware that an active transaction may read an object's old version. The code must be written to take this fact into account.
- As a result of the previous point, only transactions that modify objects need to be validated at the end of the execution phase. Of course, each frontend needs to maintain sufficient information about recently committed transactions so that it can validate later transactions.
- Forward validation schemes do not validate the read set of a committing transaction T. They just need to certify T's write set (usually small) whereas backward validation schemes also need to validate T's read set (which is potentially large).
- In case of a conflict, forward validation schemes provide the flexibility of aborting the active or the validating transaction whereas backward validation schemes always abort the active transaction.

However, forward validation schemes have some drawbacks which led us to choose backward validation for our design:

- In a system like Thor, forward validation can be viewed as a strategy in which the participants of a committing transaction T are not only the affected servers but also the frontends that have cached objects in T.MOS. Making frontends be participants of a transaction is not desirable because increasing the number of participants in the commit protocol increases the load of the coordinator server. Another problem with this approach is that frontends are not highly available; this can unnecessarily delay the commit process or abort a validating transaction.
- Forward validation schemes assume that each active transaction's read/write sets are known during T's validation phase. In a distributed system like Thor, this requires extra communication during the execution or validation phase. Furthermore, active transactions have to be blocked while a distributed validation is being carried out for T — an expensive proposition.
- Pure forward validation schemes abort committing transactions in favor of active transactions that may abort ultimately. On the other hand, backward validation schemes abort active transactions in favor of already committed or committing transactions.
- Livelock among transactions can occur if forward validation is being used whereas the backward validation schemes can avoid such situations.

3.2.1 Total Ordering of Transactions

Serializability means that the effect of executing the transactions concurrently is the same as the effect of executing them in some serial order. This serial order is called the *equivalent serial schedule*. The concurrency control mechanism can guarantee serializability in a distributed system by scheduling conflicting transactions in the same relative order at all sites. Pessimistic schemes such as 2-phase locking [Bern87] achieve this affect by locking objects and delaying transactions that try to execute conflicting operations. Most distributed optimistic schemes use timestamps to ensure that transactions at different sites are committed in the same relative order.

Our design uses globally unique timestamps to guarantee serializability. The timestamp of each committed transaction T can be viewed as the time when T executed in an equivalent serial schedule H. That is, if T had been executed at time T.ts (instantaneously), it would have read the same values as it did while running concurrently with other transactions. The timestamp is a predictor of the commit order for validating transactions; the coordinator predicts T's position in H and sends prepare messages to the participants. The validation algorithm checks whether the serial schedule H made up of all committed transactions placed in timestamp order will *remain* an equivalent serial schedule if T is inserted in H at a place determined by T.ts. The TM also has to ensure that committing T will not violate any serializability constraints with respect to the prepared transactions. That is, it has to check that none of T's operations have been invalidated by a prepared transaction; since backward validation is being used, T is not validated against active transactions. T validates successfully against a committed or prepared transaction S only if the timestamp order is the same as the commit order, *i.e.,* if S.ts is less than T.ts, then T passes validation only if S can precede T in H and vice-versa.

It is important that the coordinator chooses an appropriate timestamp value for a validating transaction T, since otherwise some transactions may be aborted unnecessarily. For example, if T has read x installed by S, T.ts must be greater than S.ts. If the coordinator chooses a low value for T.ts, T may not pass this test. Similar problems occur for later transactions if the coordinator chooses a high value for T.ts. Timestamp values are also important for validating T against concurrently committing transactions. For example, if S is modifying x and T has read it, an excessively low value of S.ts can abort T unnecessarily.

Various techniques for choosing timestamps have been suggested in the literature. For example, the scheme described in [Agarwal87] requires each site to maintain a monotonically increasing counter that is updated according to incoming commit messages. But this scheme can lead to unnecessary aborts or retries because the counters at different servers are not updated at the same rate. A negotiation-based scheme has been suggested in [Sinha85], but it suffers from the disadvantage of an extra network roundtrip delay on the critical path.

Essentially, these schemes synchronize the counters at various sites as part of the concurrency control mechanism. The Thor model assumes that loosely synchronized clocks [Lamport78] are available in the system; this is a reasonable assumption for current systems where protocols such as the Network Time Protocol [Mills88] provide such a facility. To choose a timestamp for transaction T, the coordinator server uses its local clock and augments it with the server id to make T.ts globally unique. Since loosely synchronized

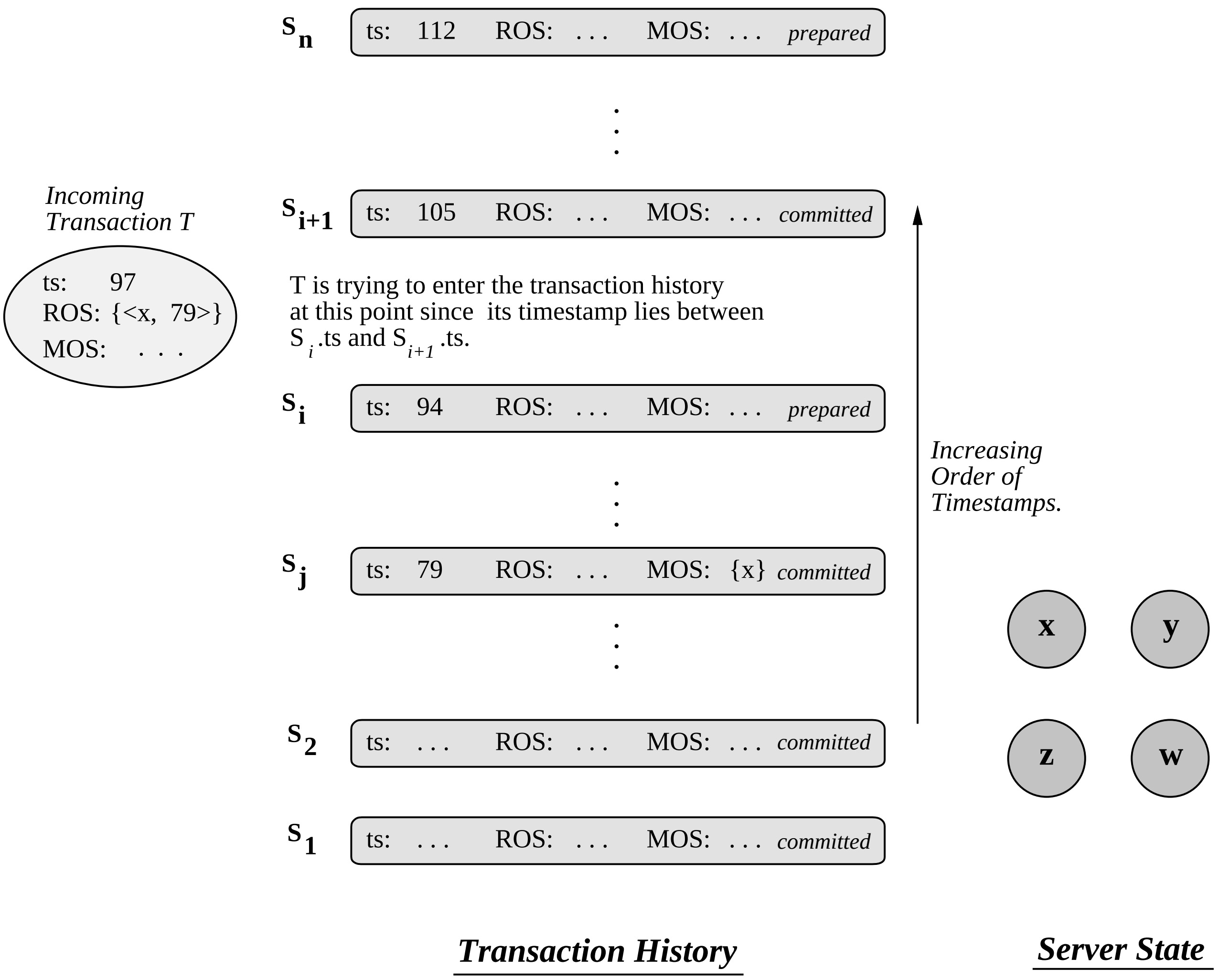


Figure 3-2: Validation using transaction history

clocks are close to real time at all sites, it is likely that if S and T are assigned timestamps (in that order) at sites A and B, then S.ts < T.ts. In general, the timestamp of an incoming transaction will usually be greater than the timestamp of a conflicting committed/prepared transaction. Thus, there will be few aborts due to excessively low or high timestamp values. Being close to real time, loosely synchronized clocks also reduce the likelihood of external consistency [Gifford83] being violated. A violation of external consistency occurs when the ordering of operations inside a system does not agree with the order a user expects. For example, if two users commit transactions S and T (in that order), they would expect S's updates to be installed before T's updates or S to commit and T to abort if they cannot be serialized in that order.

3.2.2 Validation Using the Transaction History

In this section, we develop the validation algorithm using an approach similar to the ones discussed in [Gruber89] and [Agarwal87]. We do not discuss issues such as logging in this section; these issues are addressed in the next two sections; in Sections 3.3 and 3.4, we suggest ways of modifying the scheme to make it more practical for implementation purposes.

The TM maintains sufficient validation information about prepared and committed transactions by keeping the complete history of committed and prepared transactions sorted by timestamp order. For each prepared or committed transaction S_i, it keeps an entry in the history list with the following attributes — MOS, ROS, ts and a boolean that indicates whether S_i is prepared or committed. When transaction T reads object x, a tuple of the form $\langle$*x, install_ts*$\rangle$ is inserted in T.ROS; *install_ts* is the timestamp of the transaction that has installed the version of x read by T. When T modifies x, a tuple of the form $\langle$*x, newval*$\rangle$ is added to T.MOS where *newval* is the modified value of x.

Suppose a transaction T reaches the server for validation such that S_i.ts < T.ts < S_{i+1}.ts. This scenario is shown in Figure 3-2. The TM has to validate T.ROS against transactions older than T and T.MOS has to be verified against transactions younger than T. It uses the following tests to perform validation (x $\in$ T.ROS and y $\in$ T.MOS):

1. **ROS test** — This test validates the objects that have been read by T. Let S_j be the transaction from which T has read x, *i.e.,* S_j.ts is equal to the value of *install_ts* in x's ROS tuple. The TM verifies that no prepared/committed transaction in the range from S_{j+1} to S_i has modified x. This condition guarantees that T would have read the same values in an equivalent serial schedule H as it did while running concurrently. Furthermore, the TM also verifies that T.ts is greater than S_j.ts. This condition ensures that T occurs after S_j in H.

2. **MOS test** — The TM validates T.MOS by verifying that T has not modified any object y that has been read by a transaction in the range from S_{i+1} to S_n. This condition guarantees that T does not invalidate the read operations of any of these transactions.

Note that the MOS test was not needed for the scheme suggested in [Gruber89] because the transaction manager allowed T to pass validation only if T.ts was greater than the timestamp of all previously validated transactions at that site. Since we do not require transactions to be validated in increasing order of timestamps, the TM has to perform the MOS test also. If T passes both the ROS and MOS tests, the TM inserts T between S_i and S_{i+1} in the transaction history and marks it as prepared. When the TM receives a commit message from the coordinator, it installs T's updates, marks T's entry as committed and sends its acknowledgement.

3.3 Truncating the Transaction History

The last section described the validation scheme at an abstract level without considering the space or logging requirements. In this section and Section 3.4, we discuss these issues and present a validation algorithm that is practical to implement.

To cut down on space requirements, the TM needs some way of truncating the transaction history while maintaining sufficient validation information about prepared and committed transactions. Once a transaction has committed, its modifications are installed at the relevant servers. At this point, its entry can be deleted from the history list. The TM captures the ROS and MOS information of committed transactions by maintaining two attributes for each object — *rstamp* and *version*. The rstamp attribute denotes the highest timestamp among committed transactions that have read that object. The version field of object x stores the timestamp of the transaction that has installed x's current base version. As stated in Chapter 1, it may be possible to serialize more transactions by maintaining information about multiple versions of each object. However, we did not choose this approach because it increases space overheads per object and complicates the validation algorithm. Furthermore, due to low-contention on objects, we expect transactions to have read the latest versions of objects. Thus, the latest version of objects would usually be sufficient for validating a transaction's read operations. Sections 3.3.1 and 3.3.3 discuss how the version and rstamp attributes are used by the TM for validation purposes. Section 3.4 shows how to reduce the overheads of this information.

For prepared transactions, the TM maintains a data structure called the *Validation Queue* or VQ; this idea has been suggested in [Gruber89]. This queue contains an entry for each prepared transaction S and each entry has the following attributes — ROS, MOS and ts. Essentially, the VQ just contains those entries in the history list that were marked as prepared. An update-at-site transaction T is entered in the VQ if it passes validation. It is removed from the VQ after its updates have been installed. Thus, entries are added at the end of the VQ but not necessarily removed in first-in-first-out order. If R is a read-only-at-site transaction at a server and it passes validation, it is assumed to have been committed; after updating the rstamp attributes of the R.ROS objects, the TM removes R from the VQ.

The TM uses the VQ and the version/rstamp attributes to perform the ROS and MOS tests for an incoming transaction. For ease of presentation and understanding we partition the algorithm into two parts — validation against committed transactions and validation against prepared transactions. The former tests are performed by the *Version Check* and *Rstamp Check* whereas the incoming transaction is validated against prepared transactions using the *Validation Queue Check* (VQ-*Check*).

3.3.1 Validating the ROS Against Committed Transactions

The version field of object x truncates modification history of x. It only maintains the timestamp of the transaction that has installed x's current base version. As a result, the TM does not have information about older versions of x; it must abort any incoming transaction that has read an older version of x.

To perform the version check, the TM verifies that an incoming transaction T has read the current base version of each object x in T.ROS, *i.e.,* x.version must the same as x_{base}.version. This part of the version check ensures that x has not been modified since T read it; the server state with respect to objects in T.ROS has remained the same since T read those objects. The TM also needs to check that T occurs after all transactions from which it has read objects. The pseudo-code for the version check is shown in Figure 3-3.

% Version Check
for each object x in T.ROS do
    if (T.ts < x_{base}.version or x.version $\neq$ x_{base}.version) then
        signal "Abort T".

Figure 3-3: The version check

Figure 3-4 shows a scenario in which an incoming transaction T has read object x and y from an server. But before T validates, another transaction S installs a new value of object x (newx is the new value of x that S installs) and changes the version field value from 65 to 86. T passes the version check on object y but fails this test on object x and aborts.

3.3.2 Validation Against Prepared Transactions

If an incoming transaction T passes the version check, the TM validates T against the set of prepared transactions using the *Validation Queue Check* or the VQ-*check*. If any prepared transaction S is invalidating T's operations, the TM can either wait for S's outcome to be known or abort T. The former scheme has the disadvantage that it can result in a deadlock. Our design uses the latter strategy since we assume that prepared transactions are likely to commit.

The TM performs the ROS test for T against prepared transactions that have a timestamp value less than T.ts. To pass the ROS test, T should not have read any object being modified by such a transaction. Similarly, to pass the MOS test, T should not modify any object that has been read by a prepared transaction whose timestamp is greater than T.ts. Thus, T

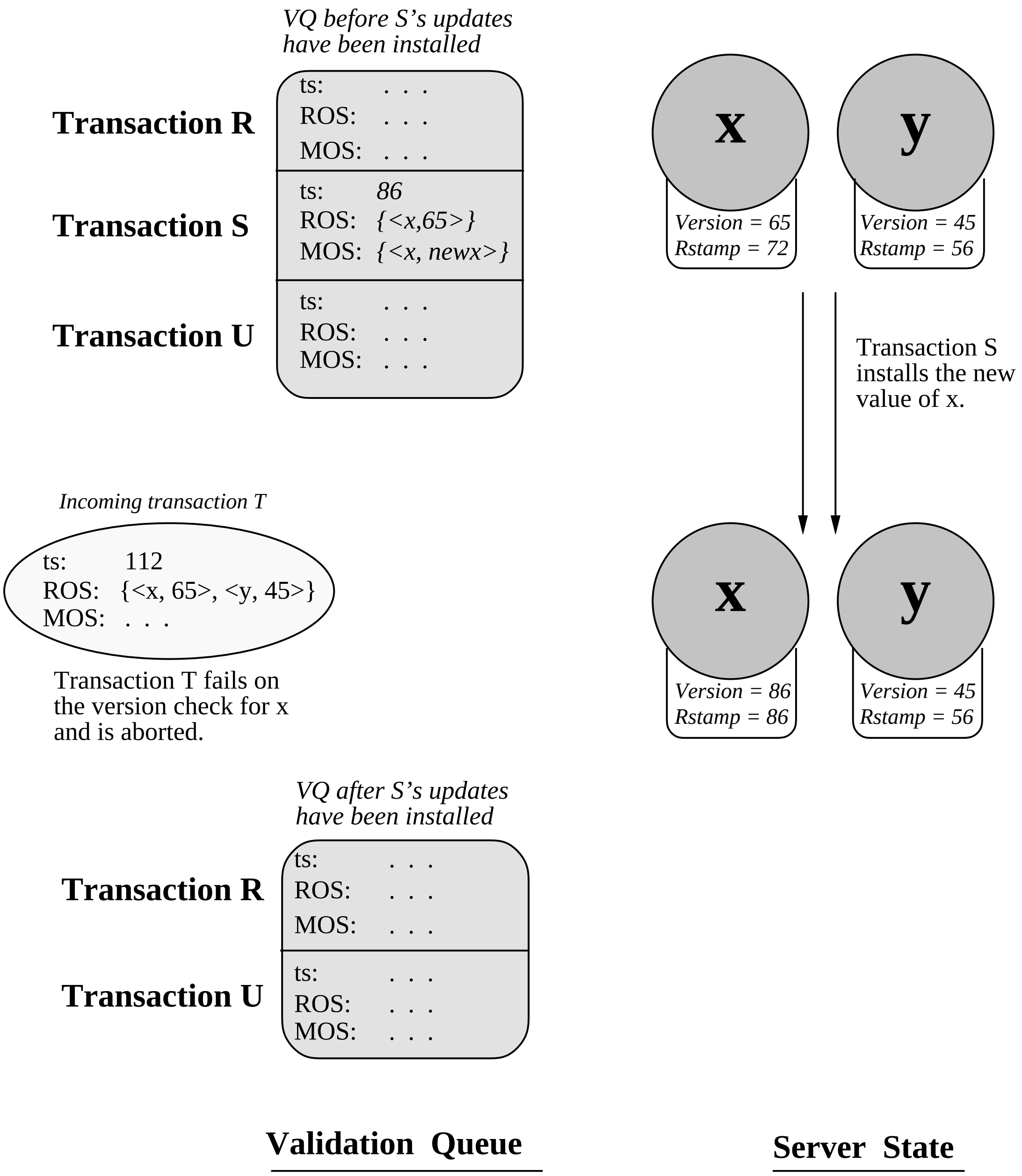


Figure 3-4: Validation failure on the version check

passes the VQ-check if the following conditions are satisfied (S is a prepared transaction in the VQ):

1. If S.ts < T.ts, then S.MOS $\cap$ T.ROS = ϕ <u>ROS test</u>

2. If S.ts > T.ts, then T.MOS $\cap$ S.ROS = ϕ <u>MOS test</u>

If any of T's reads or writes has been invalidated by a prepared transaction, T fails this check. In Figure 3-4, if transaction T reaches the server before S's updates have been installed, it will pass the version check, but it will fail the ROS test of the VQ-check and abort.

If T passes all validation checks, the TM inserts it in the VQ and sends a yes vote to the coordinator. On receiving the commit decision from the coordinator, the TM installs T's modifications and updates the version field of the relevant objects. It also removes T's entry from the VQ. Thus, at any instant of time, there is at most one transaction in the VQ that is trying to modify a particular object (since the TM does not allow blind writes). In other words, there is at most one potential version of an object in the VQ.

3.3.3 Validating the MOS Against Committed Transactions

In this section, we motivate the need for the rstamp attribute and discuss how the TM performs the MOS test against committed transactions. Suppose that the TM does not maintain the rstamp attribute for each object. After committing a transaction, it updates the version field and discards the ROS information. As the following example demonstrates, lack of read timestamp information can lead to non-serializable conditions (assume that transactions S and T are executing concurrently at different frontends):

1. Transaction S has read object x_p from site A and is modifying y_m at site B. It passes validation at both servers and S.ROS is removed from site A.

2. Transaction T, with T.ts < S.ts, has read version y_m and is modifying version x_p. It passes validation at site A because S.ROS has been removed from the VQ. It validates successfully at site B also since S's updates have not been installed at that site. S is still in the VQ and according to the ROS test, the TM at server B serializes T before S.

Both transactions pass validation and commit. However, these transactions are non-serializable and at least one of them should have been aborted. In the current scenario, at site B, S and T have committed in timestamp order but at server A they have committed in opposite order. The TM at site A should have prevented T from validating since T's write on x had been invalidated by S's read. Therefore, whenever a transaction's ROS is removed from the VQ, some information must be maintained to prevent transactions such as T from committing and violating the commit timestamp order. The TM achieves this effect by maintaining the rstamp attribute for each object. To validate T.MOS against committed transactions, the TM checks that T.ts is greater than x.rstamp for each object x in T.MOS. This test, called the *Rstamp Check*, guarantees that T's modifications have not been invalidated by read operations executed by committed transactions.

The rstamp and version attributes of relevant objects are updated when a transaction is removed from the VQ. For a read-only-at-site transaction, the rstamp attribute is updated after the transaction has passed validation.

3.3.4 Failure of Transaction Validation

If a transaction fails validation, the simplest strategy is to abort it. Instead, if possible, the TM may take steps to avoid aborting the transaction. If an incoming transaction T has a low timestamp value, the TM can ask the coordinator to raise it. For example, if T fails the Rstamp test or MOS-test of the VQ-check, the TM can ask the value of T.ts to be increased. Thus, at the cost of some extra foreground messages, the TM may be able to commit T. However, in our design, a participant never asks a coordinator to lower T's timestamp value (*e.g.,* if T fails the ROS-test of the VQ-check due to a transaction S in the VQ); instead, it aborts T. Retrying T with a lower timestamp may not be helpful because transaction S would have probably committed by the time T's retry message reaches the server. Therefore, on retry, T would fail the watermark check and abort. A retry with a lower timestamp can also lead to a ping-pong effect, *i.e.,* T.ts might be increased and decreased alternatively. Furthermore, decreasing transaction timestamps can cause timestamp values to drift away from real time.

3.4 Reducing the Space and Logging Overheads

This section explores techniques to reduce the per object space requirements for concurrency control purposes, *i.e.,* decrease the space overhead for the version and rstamp attributes. It also suggests a strategy to avoid logging information at a read-only participant.

3.4.1 Maintaining the Read History Information

Suppose that U is a read-only-at-site transaction at site A. If U passes validation, the TM sends its yes vote but does not need the commit decision from the coordinator. However, if U.ts is greater than the rstamp attribute of an object x in U.ROS, the TM has to update x.rstamp to be U.ts (which will normally be the case because of loosely synchronized clocks). The server must flush the new rstamp information to stable storage before replying to the coordinator. Therefore, not only does the rstamp attribute have a space overhead, it also requires a foreground log flush for read-only-at-site transactions. The rstamp attribute problem can be alleviated in the following way:

Instead of maintaining the rstamp attribute per object, the TM *approximates* this information by storing an upper bound on the read times of all objects at a server. This bound is called the *read-watermark* or X_r for the server. Since the TM has lost the read information for each object, it must assume that all objects at the server were read at time X_r. Thus, to ensure that an update-at-site transaction T does not invalidate a read operation of a committed transaction, the TM has to verify that T.ts is greater than X_r. This test is called the *Read-watermark Check*. However, lack of per-object information can cause spurious aborts. But we expect such aborts to be rare since loosely synchronized clocks are being used to generate timestamps; transactions reaching a server will usually have timestamp values greater than the read-watermark at that server (see Section 3.2.1).

When S.ROS is removed from the VQ, X_r is set to the maximum of its original value and S.ts. In our design, we remove a transaction S from the VQ after it has committed (Section 3.4.3 discusses other possibilities). If S is a read-only-at-site transaction, S.ROS is removed after S validates successfully at the site. If S.ts is greater than the read-watermark, the TM must force the new value of X_r to the backup. This is likely to happen because loosely synchronized clocks are being used for generating timestamps; S.ts will usually be greater than the last transaction committed at that site. Thus, the read-watermark approach avoids space overhead per object, but read-only participants still require a log record to be flushed on the critical path.

The TM can avoid updating X_r at a read-only participant using a technique suggested in [Liskov91]. The TM maintains a stable copy of the read-watermark called the *stable-read-watermark* or X_{rs} with the invariant: $X_{rs} \geq X_r$. The stable read-watermark is a value maintained on stable storage. X_r is only maintained in memory; it is not kept on stable storage. Initially, X_{rs} is set to $X_r + \delta$ (the choice of δ is discussed in the next paragraph). Whenever a transaction prepare increases the value of X_r above X_{rs}, the latter is set to $X_r + \delta$ and forced to the backup. Thus, a read-only participant avoids a roundtrip delay on the critical path if the new value of X_r is below X_{rs}. As an optimization, the TM can update X_{rs} to $X_r + \delta$ when an update-at-site transaction prepares or commits since it has to force a prepare/commit record to the backup anyway.

If a site crashes and recovers, the read-watermark is initialized to the stable-read-watermark. This step ensures that the invariant $X_{rs} \geq X_r$ is still maintained and X_r is always an upper bound on the read times of objects at a server. However, an update-at-site transaction with a timestamp greater than the original value of X_r but lower than the new value may fail the watermark test and be unnecessarily aborted. A large value of δ increases the likelihood of such aborts. This seems to suggest that δ must be kept small. But a low value of δ will force frequent updates of X_{rs}, defeating the primary purpose of the stable-read-watermark. The choice of δ can be made based on system characteristics. For example, in a system like Thor where messages are periodically exchanged between the primary and the backup every t_a seconds, δ can be set to t_a and the new value of X_{rs} piggybacked over the liveness messages. If t_a is sufficiently less then the time taken to complete a view change [Oki88], the probability of a transaction being aborted due to the reinitialization of X_r with X_{rs} becomes insignificant.

3.4.2 Implementing the Version Field

The preceding discussion assumed that an object's version field stores the timestamp of the transaction that installed the latest version. Let us analyze the space overhead of using timestamps for the version field. As stated earlier, a globally unique timestamp can be implemented by augmenting a server's local timestamp with the server identification number. Suppose that the local timestamp and the server id require n and d bits of storage respectively. Thus, the version field has a space overhead of $n + d$ bits per object. Another way of implementing this attribute is to use a k-bit counter (k < n+d), *i.e.,* every object update causes the version field to be incremented by 1. The version field implemented using a counter is also referred to as the object's *version number* or *vnum*.

Version numbers not only use less space, they also wrap around at a slower rate compared to timestamps. Wrapping around of the version field for the counter scheme depends on the rate at which an object is modified; for the timestamp implementation it depends on the clock rate. Suppose that a counter implementation uses 32 bits and a timestamp scheme take 64 bits of storage (32 bits for local time and 32 bits for the server id) to implement the version field. Assume that an object is modified 100 times per second and the granularity of the clock is 1 millisecond. The counter value will wrap around after 1.3 years whereas the timestamp value will wrap around after 50 days. Timestamp values run out faster because only 1 in 10 clock values is being used for a new version value; the remaining values are "wasted". Wrapping around of version numbers can cause serializability problems; this issue is discussed later in this section.

It might seem that the modification history is accurately captured by a counter implementation; any transaction T that has read an old version or has an inappropriate timestamp value will be aborted by the version check. But this is not true since the counter value does not capture the timestamp of a modifying transaction S. This information is lost when S.MOS is removed from the VQ. The TM can perform only part of the the version check; it cannot ascertain that T.ts is greater than the timestamp of all transactions from which T has read objects. Here is an example to show how non-serializable behavior can occur if the version field is implemented using counters.

1. Suppose that S is a transaction modifying object x_p at site A and y_m at site B. S commits and passes the validation checks at A and B. The new version of x (x_{p+1}) gets installed at A.

2. Another transaction T that has read x_{p+1} and y_m, tries to commit with a timestamp less than S.ts. T passes the version check (because it has read x's new version) and the VQ check (because the VQ no longer contains S.MOS) at site A. Since T.ts < S.ts and S is still in the VQ at site B, T succeeds in validating at site B also.

But S and T are non-serializable and at least one of them must abort. The TM at site A must abort T since T is trying to serialize in an order that is different from the timestamp order; T.ts is less than the timestamp of the transaction (S) from which it has read object x. The validation mechanism failed to detect this behavior at site A because the version field of x did not have sufficient information to abort T. Thus, the TM needs to maintain some information about S's timestamp when it installs S's updates and removes S from the VQ.

Maintaining the Write Timestamp Information Using a Write-watermark

Instead of keeping track of the individual installation times of objects, we can use an approach similar to one used for the rstamp attribute. The TM use a counter for the version field and maintains an upper bound on the modification times of all objects at the server. This bound is called the *write-watermark* or X_w. Since the TM has lost the fine granularity per-object update information, it must assume that all objects at the server were modified at time X_w. A transaction validating at that server must have a timestamp greater than X_w; otherwise it is aborted. This test is called the *Write-watermark Check*. Note that unlike the rstamp case, the TM still needs to maintain the version field for each object. This is so because an upper bound on the write times is not sufficient to check whether object x has been modified since T read x.

The write-watermark captures the modification time history of all objects in a compact way. When the TM removes S.MOS from the VQ, it updates X_w to be the maximum of the original value and S.ts. In our design, the S.MOS is removed from the VQ after installing S's updates. Section 3.4.3 discusses other options.

Timestamps or counters for the version field?

The timestamp implementation of the version field has the advantage that no transaction is aborted due to lack of per-object write timestamp information and no write-watermark has to be maintained. But as stated earlier, timestamps have a higher space overhead than counters. Since it is unlikely that a transaction will fail the write-watermark test, increasing the space overhead per object to avoid a rare class of aborts is not a good design decision. For the rest of the thesis, we assume that the version field is implemented using counters; the TM validates the ROS of an incoming transaction T against committed transactions using the version and the watermark checks.

Thor is intended to be used over a long period of time and it may seem that wrapping around of counters may cause problems. However, this is not the case since version numbers are used just to check if the validating transaction T has read the latest version of its ROS objects or not, *i.e.,* version numbers are used for an equality test not for ordering purposes. To determine whether T's timestamp value is in consonance with its position in the transaction history (with respect to committed transactions that have modified the T.ROS objects), the write watermark is used. Thus, it is the wrapping around of watermarks and not the version numbers that is an important consideration. To prevent watermarks from wrapping around during the lifetime of a database, they can be implemented as large-sized timestamps. Note that these watermarks are just being maintained per server and not per object; as a result, their space overhead is insignificant. Assuming a 1 microsecond clock, watermarks implemented using 88 bits will last for more than 2000 years.

However, there can be serializability problems if a frontend has an object x_v in its cache and x_v's version number v is re-used by the relevant server. We assume that such situations will not occur; a frontend will receive invalidation messages before the version number of some object in its cache is reused by the relevant server. This is a reasonable assumption because the time it takes for even a small vnum (*e.g.,* 16 bits) to wraparound is relatively large compared to the time taken for a frontend to be informed about an invalidation. However, if an object's version number is reused and a frontend F still has the object's old copy, the relevant server can send a message to kill F.

3.4.3 Updating the Watermarks

As discussed in the previous sections, we remove a transaction's entry from the VQ after it has committed and update the watermarks. However, it is not necessary to update X_r or X_w at commit time. The watermarks can be updated at any stage during (or after) a transaction's lifetime. We will consider two possibilities for updating the watermarks:

1. The transaction manager updates the watermarks after preparing an incoming transaction S. For the read-watermark, this scheme has the advantage that S.ROS does not have to be kept in the VQ and the TM does not have to validate the MOS of an incoming transaction T against S.ROS; if T has passed the read-watermark test, its timestamp must be greater than S.ts. However, this scheme essentially constrains transactions to arrive at a server in increasing order of timestamps which may result in spurious aborts. But as stated in Section 3.2.1, it is likely that transactions arrive at a server in increasing order of timestamps.

2. The TM keeps a transaction's entry in the VQ even after that transaction has committed at that site. The VQ check has to be modified for this approach since the VQ may now contain different versions of the same object. This scheme can be used by a TM if it discovers that there are excessive aborts due to transactions failing (say) the write-watermark check. Note that this situation is unlikely in normal circumstances but may occur in some cases, *e.g.,* if the clock skew is high. The TM at server A could wait to remove the entry of transaction S from the VQ until the following condition is satisfied:

   $$S.ts < C_t - \alpha - \epsilon - \beta \qquad \textit{Removal Condition}$$

   where C_t is the current time at site A, ϵ is an estimate of the clock skew, α is the network delay and β takes retries into account.

   If the local time at site A is C_t, then (most likely) the local time at any other site is greater than $C_t - \epsilon$. No messages that were sent more than α seconds earlier will reach site A. We also assume that when a server's current time is G, the earliest prepare message it can send at this time cannot have a timestamp value less than $G - \beta$. This implies that at time $C_t - \alpha - \epsilon$, the lowest timestamp that could have been assigned to a transaction by a server is $C_t - \alpha - \epsilon - \beta$. Therefore, the TM can remove the entry of any committed transaction that has a timestamp less than this value and update the watermarks. Basically, the TM reduces the chances of an abort due to the watermark checks by retaining some entries of the transaction history for a sufficiently long time. But this approach increases the VQ's space overhead and also slows down the VQ-check since an incoming transaction has to validate against more transactions.

We use the first scheme for read-only-at-site transactions so that the TM just has to update X_r (which it may not have to if X_r is less than X_{rs}). For update-at-site transactions, we retain the ROS information until commit time since the MOS information has to be kept anyway. The second scheme complicates the validation algorithm and slows down the VQ-check since an incoming transaction has to validate against more transactions. We assume that clock skews are low and retries are rare. As a result, it is very likely that the removal condition will be satisfied at the time S's updates are installed. Thus, in our design, the TM removes S's VQ entry information after S has committed.

3.5 The Serial Validation Algorithm

This section presents the basic validation algorithm for an incoming transaction. The algorithm is serial, *i.e.,* at most one validation or installation can be happening at a time[2]. A synchronization lock called the VQ-*lock* is used for this purpose; the TM holds this lock during the validation process and while installing a transaction's updates. We also assume that a synchronization lock is available for each object. The TM acquires a write-lock on an object x, modifies x and then releases the lock; therefore, it holds only one object write-lock at at a time. A read-lock is acquired on an object by operations such as object fetch and version check before reading the object. Note that deadlock cannot occur between installation and validation because these operations acquire the VQ-lock before proceeding with their work. Furthermore, installation cannot deadlock with object fetch (which may acquire multiple read-locks) because it holds at most one object lock at any given time. Before discussing the algorithm, let us summarize the design decisions that have been made for the transaction manager:

1. The version field is implemented using a counter. A transaction's MOS is removed from the VQ after installing its modifications and the write-watermark X_w is updated.

2. A transaction's ROS is removed from the VQ after the transaction has committed at that site and its updates have been installed. For a read-only-at-site transaction T, the TM removes the T.ROS after T has passed validation. As stated in Section 3.4.3, if this causes too many aborts, the TM can retain the information for a longer time. The read-watermark X_r and its stable version X_{rs} are maintained using the technique described in Section 3.4.1. Since Thor does not permit blind writes, X_r is always greater than or equal to X_w. For the sake of simplicity, the algorithm discussed in Figure 3-5 assumes that the stable-read-watermark technique is not being used, *i.e.,* X_r is flushed to the backup whenever its value increases.

Figure 3-5 gives the details of the validation algorithm. In the figure, we assume that all locks held by the TM are released when an signal is raised, *e.g.,* when a transaction fails the version check on object x, the TM releases the VQ-lock and the read-lock on object x and

[2] A parallel validation scheme that allows transaction validations and object installations to proceed simultaneously is discussed in the next chapter.

T Enters Validation
Lock (VQ*-lock)*

Write-watermark Check
if (T.ts $<$ X_w) then
ask the coordinator to retry with a timestamp greater than X_w.

Read-watermark Check
if (T.ts $<$ X_r and T.MOS $\neq \phi$) then
% A non-read-only-at-site transaction has failed the read-watermark test.
ask the coordinator to retry with a timestamp greater than X_r.

Version Check
for each object x in T.ROS do
Read-lock(x)
if x.vnum $\neq$ x_{base}.vnum then signal "Abort T".
Unlock(x)

Validation Queue Check
ROS test: If (S.ts $<$ T.ts and S.MOS $\cap$ T.ROS $\neq \phi$) then
signal "Abort T".
MOS test: If (S.ts $>$ T.ts and T.MOS $\cap$ S.ROS $\neq \phi$) then
ask the coordinator to raise T's timestamp.

Validation Succeeded
If T.MOS $= \phi$ then
X_r := max (X_r, T.ts)
else insert_in_queue (VQ, T) *% Add T to the* VQ.
Unlock (VQ*-lock)*
% Send Yes vote to the coordinator and wait for the decision.

Installation
Lock (VQ*-lock)*
If decision is "commit" then
for each object x in T.MOS do
Write-lock(x)
x_{base}.vnum := x.vnum
x_{base} := x *% Install the new version of x.*
Unlock(x)
X_r = max (X_r, T.ts); X_w = max (X_w, T.ts)
Remove T from the VQ.
Unlock (VQ*-lock)*

Figure 3-5: The serial validation algorithm then signals abort. In the VQ-check, the ROS test ensures that an earlier transaction has not invalidated T's read whereas the MOS test ensures that T is not modifying an object that has been read by a later transaction. As described in Section 3.3.4, the TM asks the coordinator to retry with a higher timestamp in MOS test but aborts T in the other case.

In the MOS test, before asking the coordinator to retry T with a higher timestamp value *new_ts*, the participant can check whether there is another transaction U in the VQ that would demand a retry with an even higher timestamp. Otherwise, if T is retried with *new_ts*, U would force T.ts to be raised again. To avoid such multiple retries, the participant can validate T against all transactions in the VQ (unless T fails validation) and then decide whether it should signal a retry or an abort. It can maintain a lower and upper bound on the permissible values for T.ts called *min_val* and *max_val* respectively. During the VQ check, if this range becomes empty, T is aborted. Furthermore, if some transaction S is restricting the maximum value of T.ts, the TM can signal abort instead of retry. The TM can make this decision on the assumption that it will receive the revalidation message for T (with a new timestamp less than max_val) after S has committed; committing S will raise the read/write watermark level and cause T to fail the watermark check.

If T is a single-server transaction the coordinator need not assign a value to T.ts before the VQ check. After completing the VQ check it can select any value in the range [min_val, max_val]. The TM can make a choice according to T's characteristics. For example, if T is predominantly read-only, the TM can select min_val for T.ts since this value would permit a higher range of timestamp values for later transactions that modify objects in T.ROS. Similarly, if T has modified many objects, the TM can choose max_val for T.ts. However, assigning min_val or max_val to T.ts may result in loss of external consistency since the TM may choose a timestamp far from the current time. Thus, in our design, we have not adopted this strategy.

In this chapter, we presented a strategy in which there is some space overhead per object for concurrency control. We have developed a scheme in which there is no space overhead per object; see [Adya94] for details. In that paper, we have also discussed a validation algorithm in which a transaction's VQ entry may be retained in the VQ even after the transaction has committed.

Chapter 4

Optimizations

This chapter explores various optimizations that can be used in Thor to minimize the delays observed by an application. Section 4.1 presents a strategy to allow multiple validations and installations to proceed in parallel at a server. Section 4.2 presents more optimizations to reduce the foreground delays of the commit protocol. Section 4.3 discusses a scheme in which the application does not wait for the current transaction's outcome to be known before starting the next transaction.

4.1 Parallel Validation and Installation

The last chapter discussed the basic strategy used in Thor for validating an incoming transaction. However, it required that at most one transaction validation or installation of objects could be happening at any given time. This section presents a scheme to allow the validation of transactions and installation of objects to proceed simultaneously at a server. The algorithm described in this section is based on the assumption that there are synchronization locks available for each object and for the VQ/watermarks (VQ-lock).

4.1.1 Parallel Validation

If two conflicting transactions S and T validate at a server, the TM needs to ensure that at least one of them detects the conflict and takes appropriate action. Let us call this requirement the *conflict-detection property*. To maintain this property and permit multiple validations at a server, we use a strategy similar to the one suggested in [Gruber89]. For an incoming transaction T, the idea is to take a fast snapshot (inside a critical section) of all the relevant activities that can occur during T's validation process and then perform T's validation using the snapshot. To generate this snapshot, the TM locks the VQ, makes a copy called the VQ-*copy* and enters T in the VQ. It performs the watermark checks and releases the VQ-lock. For the version check, the TM read-locks a ROS object, performs the test and releases the lock. It performs T's VQ-check using the VQ-copy, *i.e.,* it assumes that all transactions in VQ-copy are prepared and validates T against them. It is possible that some of the validating transactions in VQ-copy that cause T to abort get aborted themselves, *i.e.,*

<u>T Enters Validation (Watermark Checks)</u>
Lock (VQ*-lock)*
Generate VQ-copy from VQ. Enter T in the VQ.
Perform the write-watermark test.
Perform the read-watermark test.
Unlock (VQ*-lock)*

<u>Version Check</u>
for each obj, x, in T.ROS do
Read-lock(x)
Perform the version check for x.
Unlock(x)

<u>Validation Queue Check</u>
Perform the VQ-check using VQ-copy.

% Validation completed.
Lock (VQ*-lock)*
If T has passed validation, mark T's entry in the VQ as *prepared.*
else remove T's entry from the VQ.
Unlock (VQ*-lock)*
Send appropriate message to the coordinator and wait for the decision.

<u>"Commit" or "Abort" decision received from coordinator</u>
Lock (VQ*-lock)*
if the coordinator's decision is commit then
Install T's updates by write-locking one object at a time.
Update the watermarks.
Remove T from the VQ.
Unlock (VQ*-lock)*

Figure 4-1: The parallel validation algorithm they fail validation at this server. In the serial validation case, T would have been validated after these earlier transactions had been prepared/aborted and it might have been possible that due to some of these aborts, T would have passed validation. However, since aborts are rare, it is likely that the validating transactions in the T's VQ-copy will pass validation.

Thus, by taking a snapshot of the VQ and making the above assumption, the TM is able to permit multiple transactions to validate simultaneously at a server. However, to prevent non-serializable behavior from occurring, object modifications are not performed while a transaction is validating. This condition is relaxed in Section 4.1.2.

Figure 4-1 shows the validation algorithm in which multiple transactions can validate simultaneously at a server. Deadlocks cannot occur between various operations because the the TM acquires at most one object lock at a time. Non-serializable conditions due to simultaneous validations cannot occur because the TM maintains the conflict-detection property. In this algorithm, if two conflicting transactions validate simultaneously, the transaction that enters the VQ later will detect the conflict. For example, suppose that S has read x and T is modifying x. Transaction S enters the VQ before T and both transactions validate in parallel. Thus, T's VQ-copy contains S's entry and T passes validation only if T.ts is greater than S.ts. In general, if two concurrently validating transactions at a server are non-serializable, the transaction that enters the VQ later will fail validation and abort.

In Figure 4-1 objects are updated atomically with respect to the addition of transaction entries in the VQ. As a result, if a validating transaction is not serializable with a transaction that is installing its updates, the former is aborted. Suppose R is a committed transaction that is installing a new version x_{p+1}, and U is a validating transaction that has read x_p. If U makes the VQ-copy after R has been removed from the VQ, U aborts on the version check. Otherwise, the VQ-copy has sufficient information to abort U if it needs to be. Thus, U is definitely aborted if it is not serializable with R.

An issue of concern in this approach is the generation of the VQ-copy. A naive implementation that copies all the VQ information is space and time expensive. Instead, the following approach can be used to achieve the effect of copying the VQ:

Assume that the VQ is implemented as an array of transaction entries. Each entry of the VQ has a counter that is initialized to zero. When an incoming transaction T needs to copy the VQ, the TM acquires the VQ-lock and records the array index of the last element in the VQ as the endpoint of T's VQ-copy. The beginning of T's VQ-copy is the first element in the VQ whose entry is marked as prepared. The TM increments this entry's counter and releases the VQ-lock. The TM carries out the VQ-check using the entries that are between these points. When T's validation has been completed, the TM decrements the counter; this step effectively destroys T's VQ-copy. If T has passed validation, the TM marks T's entry as prepared else it marks the entry as aborted. When the TM receives the coordinator's decision for a transaction, it marks that transaction's entry as committed/aborted. The TM can remove an entry from the beginning of the VQ if it is marked as committed/aborted *and* its counter value is zero. Thus, VQ-copy for a transaction can be generated without excessive space or time overheads.

4.1.2 Permitting Installations to Proceed in Parallel with Validation

We have assumed that the TM performs object installations atomically with respect to addition of entries from the VQ. In Figure 4-1, the TM achieves this effect by holding the VQ-lock while installing a transaction's updates. As a result, any incoming transaction T is delayed until the updates have been installed because the TM has to acquire the VQ-lock for generating T's VQ-copy. This delay lies on the critical path of the commit protocol and a strategy is required to reduce it.

Validation of transaction T

% As in Figure 4-1.

Installation of T's updates

```
If the coordinator's decision is "commit" then
    Lock (VQ-lock)
        Update the watermarks.
    Unlock (VQ-lock)
    for each object, x, in T.MOS do
        Write-lock(x)
            Install the new version of x.
        Unlock(x)
Lock (VQ-lock)
    Remove T's entry from the VQ.
Unlock (VQ-lock)
```

Figure 4-2: Concurrent installation of object updates

This problem can be alleviated if the TM holds the VQ-lock for only a short period of time. It holds this lock only while updating the watermarks and the VQ but not while modifying the objects. The steps executed to install a transaction's updates are shown in Figure 4-2. When the TM receives the coordinator's commit decision, it acquires the VQ-lock, updates the watermarks and releases the lock. It installs the object updates by locking the relevant objects one at a time. Finally, it acquires the VQ-lock, removes the transaction's entry[1] and releases the lock. In this strategy, it is important that the watermarks are updated before the objects are modified, as the following example shows:

Suppose the watermarks are updated after the new versions of objects have been installed. Consider the scenario in which a prepared transaction S is modifying x_p and y_q. Object x_{p+1} has been installed but not y_{q+1}. Transaction T reads x_{p+1} and y_q. It reaches

[1] In the VQ-copy implementation described in Section 4.1.1, the TM just marks the transaction's entry as committed.

the server for validation with T.ts < S.ts. It passes the watermark (X_r, X_w have not been updated till now), version, VQ checks and commits. However, T is not serializable with S and must abort.

Updating the watermarks before modifying the objects maintains the invariant that the write watermark is an upper bound on the object modification times at the server. With this strategy, transaction T in the previous example would have failed the watermark test and aborted. Thus, updating the watermarks is a critical point in the installation process.

Let us see how serializability is not sacrificed when installations are allowed to proceed in parallel with validations. We need to consider only the case in which a transaction is installing an object that a validating transaction has read. Correctness for other cases has already been discussed in Section 4.1.1; parallel installation of objects does not affect those cases. Suppose R is installing object x and U is a validating transaction that has read x. There are two cases to consider:

1. *U has read x from R.* If U is not serializable with R (*i.e.,* U.ts < R.ts), U will fail the watermark test and abort. Note that when installation was not allowed to overlap with validation (*i.e.,* the algorithm in Figure 4-1), U's validation would have been delayed till R's installation process had been completed. In the new scheme, the TM does not delay U's validation.

2. *U has not read x from R.* If R and U are non-serializable (*i.e.,* U.ts > R.ts), U will fail the VQ-check and abort. Executing R's installation process in parallel may have the benevolent side-effect of failing U before the VQ-check, *i.e.,* the watermark or version check.

Thus, in both cases, the new scheme ensures that U is aborted if R and U are non-serializable. Combining the strategy shown in Figure 4-2 with the strategy presented in Section 4.1.1, the TM can allow multiple transaction validations and object installations to proceed in parallel at a server. As the next chapter shows, this feature is very important for object migration.

4.2 More Optimizations to Reduce Foreground Delays

This section explores a few more optimizations that can help in decreasing an application's wait time during the commit protocol's first phase. In the distributed commit scheme discussed so far, the frontend has to wait for a time equal to 2 network roundtrip delays plus the time taken to perform 2 backup flushes (for the prepare and commit records). Section 4.2.1 explores a strategy to reduce network roundtrip delays by letting the frontend send the prepare messages to the coordinator/participants. Section 4.2.2 presents a scheme to avoid the backup flush of the prepare record. Section 4.2.3 discusses an optimization in which the frontend sends the transaction information to the coordinator even before the transaction has committed.

4.2.1 Short-circuited Prepare

As described earlier, when an application requests its current transaction T to be committed, the frontend chooses a coordinator server and sends a *commit-request* message that contains T's information. This server sends prepare messages to the participants and waits for their votes. Instead, the frontend can send the commit-request and prepare messages to the coordinator and participants respectively. The participants validate T's operations and send their votes directly to the designated coordinator; as before, the coordinator collects the votes and is responsible for resending validation messages. Note that this scheme requires the frontend to assign a timestamp for T. Since prepare messages are sent in parallel with the commit-request message, the foreground message delay is reduced from 4 message delays (2 network roundtrip delays) to 3 message delays. Another advantage of this approach is that it takes some load off the coordinator because the frontend is responsible for sending prepare messages to the participants.

A scenario that can happen in the short-circuited prepare scheme is that the coordinator may receive a yes vote from a participant before receiving the commit-request message from the frontend. Since there is no entry for T in the VQ, the coordinator will ask the participant to abort T (our design uses the presumed-abort strategy). This is an unlikely situation because the frontend–coordinator path involves 1 message delay whereas the frontend–participant–coordinator path involves 2 message delays plus some validation processing. The likelihood of such spurious aborts can be reduced even further if the coordinator delays its reply to the participant. That is, the coordinator compares its local clock with T's timestamp in the vote message. If T.ts is very low compared to the server's local clock, it sends an abort message to the participant. Otherwise, it waits for the commit-request message from the relevant frontend for a certain period before sending the abort message. Thus, this scheme increases the complexity of the commit protocol slightly, but it reduces foreground delays and also reduces the coordinator's load.

4.2.2 The Coordinator Log Protocol

Two log flushes are done in the first phase of the 2-phase commit protocol. These flushes are required to make the protocol resilient to crashes. But the Thor model assumes that crashes are rare. It would be preferable to develop a strategy that reduces the number of foreground log flushes for the normal case, *i.e.,* when there are no crashes. A strategy, called the *coordinator log protocol*, has been suggested in [Stamos89] to achieve this effect.

The basic idea is to avoid a synchronous log flush (forcing the record before sending the vote reply) of the prepare record by a participant. However, to ensure that the commit protocol is resilient to crashes, the commit decision must not be flushed to stable storage by the coordinator before each participant's prepare record has reached stable storage. In the normal protocol, each participant maintains its prepare record on stable storage. In the coordinator log protocol, the coordinator maintains the prepare record on behalf of each participant. Along with its yes vote, each participant sends the relevant information to the coordinator, *i.e.,* read/write watermarks, log sequence number (LSN) and other implementation-dependent log information. The LSN is sent so that the participant can order the log records during recovery. Note that the participant does not need to send the ROS, MOS and NOS information since the coordinator already has it (as discussed later in this section, this is true for the short-circuited prepare scheme also). Thus, the increase in the size of the vote message is quite small. On receiving all the yes votes, the coordinator server flushes the prepare record information along with its commit decision to its backup. This approach results in only one log flush in the foreground instead of two flushes. In Thor, where a flush involves sending the data to the backup, the application waiting time is reduced by a network roundtrip delay; in systems where the log has to be flushed to the disk, the savings are even greater.

In the coordinator log protocol, a server's log is spread over servers that have been coordinators for transactions prepared at that server. If the server crashes and recovers, it could regenerate its log by contacting all the servers in the Thor universe. But this is not an efficient way of recovering a server's log. Instead, the server keeps track of the servers that have been its coordinators in the recent past and maintains this information on stable storage in a set called RECENT-COORD. When it receives a prepare message from server A, it checks if A is a member of RECENT-COORD. If A $\notin$ RECENT-COORD, it adds A to this set and flushes it along with the prepare record. Otherwise, it just validates the incoming transaction and sends its reply to A; the prepare record is not flushed to the backup. A server can keep the the size of RECENT-COORD small by periodically removing servers that have not been the coordinators of any of its transactions in the recent past.

A problem with the coordinator log protocol is that autonomous recovery of a server (say B) has been sacrificed. This protocol requires all servers in B's RECENT-COORD to be available when B is recovering from a crash. If some of these servers are down, B's recovery will be delayed. This is not a problem in Thor where servers are highly available; the probability of B not being able to contact one of its recent coordinators is very small. But in systems where servers are not highly available, each server can maintain crash information about other servers. On receiving a prepare message from server C, a server flushes the prepare record if it considers C to be an unreliable server.

The Coordinator Log Protocol with Short-circuited Prepare

The short-circuited prepare scheme reduces the foreground delay and the coordinator log protocol reduces the number of foreground log flushes. If both these strategies are combined, the distributed commit delay of 4 message delays and 2 log flushes can be brought down to 3 message delays and 1 log flush. In Thor, the commit wait time for a frontend is reduced from 8 message delays to 5 message delays (a log flush is equivalent to a network roundtrip). Figure 4-3 shows the combination of these two strategies for committing a transaction. It describes the case where the coordinator is a member of RECENT-COORD and the prepare record is not flushed to the backup. Note that the ROS/MOS/NOS information about each participant can be sent by the frontend to the coordinator or by the participants along with their yes votes. The former strategy seems to be better because it takes the load off the participant servers.

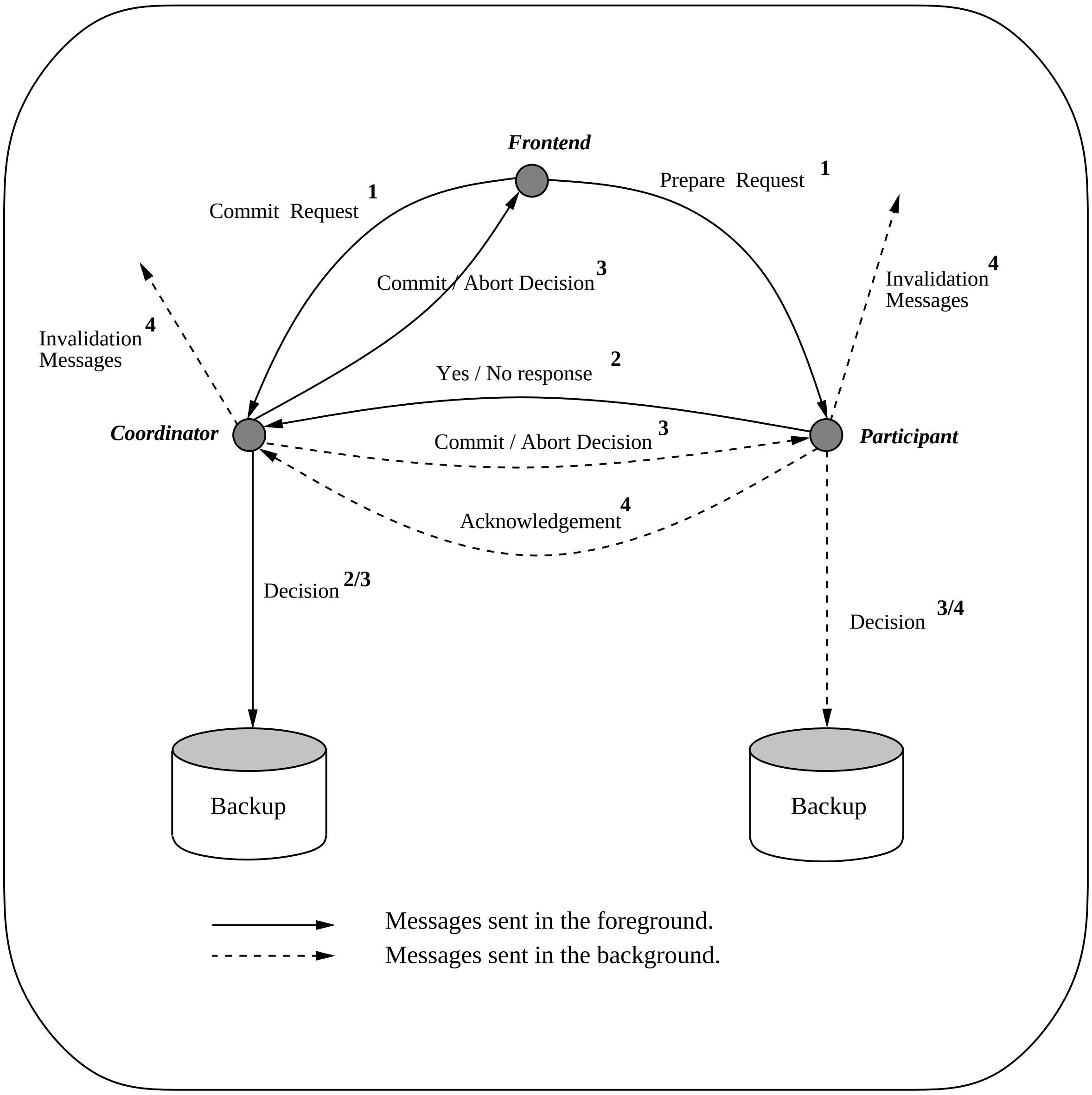


Figure 4-3: The coordinator log protocol with short-circuited prepare

4.2.3 Amortizing the Commit Cost Over a Transaction's Lifetime

A frontend sends the commit information to the servers after receiving a commit request from its application. Instead, it can send this information during the execution phase of a transaction. This scheme has the advantage that the commit time delay is amortized over a transaction's lifetime; a similar scheme has been proposed in [Oki85]. The frontend piggybacks the current transaction's read/write information on fetch requests and liveness messages. At commit time, most of the transaction information has already reached the participants and the frontend just needs to send small commit-request/prepare messages. However, this mechanism has the disadvantage that multiple updates to the same object during a transaction's execution may cause the modification information to be sent more than once to a server. This problem can be alleviated if a frontend never sends an object's update information more than n times during the execution phase, where n is a small number. Thus, an object's modification information is sent at most *n+1* times to a server — in n messages sent during the execution phase and one sent at the end of the transaction.

There are three strategies that a server can use for handling the commit information sent during the execution phase of a transaction T:

1. *Early Send* — The TM just stores this information and starts validation when it receives a message to validate T from the coordinator/frontend.
2. *Early Validation* — The TM validates T to determine whether T's operations have been invalidated by a prepared or committed transaction. This strategy still requires T to be validated at the end of the execution phase.
3. *Early Prepare* — In this case, the TM only early-validates T and adds an entry for T in the VQ if T passes the early validation process. This mechanism has the advantage that operations that have been early-validated need not be validated at T's commit time.

The early send scheme reduces the size of the commit message sent by the frontend to the coordinator and is useful in reducing the commit delay as observed by an application. The early validation scheme checks whether T's operations have been invalidated by some other prepared/committed transaction S and can prevent T from doing wasted work. But early validation is not necessary to achieve this effect; invalidation messages sent at the end of a transaction's second phase inform the relevant frontends about object modifications. If T has been invalidated by S's operations, this will be detected at the frontend when the invalidation message for S arrives from the relevant server. Thus, early validation does not offer any gains; on the contrary, it increases the load at the frontend and the servers.

In the early prepare mechanism, the timestamp used to early-validate T is also used at T's commit time. Thus, it is important that the frontend chooses an appropriate timestamp for early preparing T. Choosing the current time for T.ts has the problem that this value will be too low when T completes its execution phase and validates its remaining operations; at most of the participant servers, T will fail the watermark test and abort. Thus, validating T using the frontend's current time for T's timestamp is not beneficial and we will not consider it further.

Instead of assigning a particular value to T.ts, the frontend can assign a range of values for the timestamp value — its current time (denoted by *curtime*) to infinity. If T passes early validation for all timestamp values in this range, no transaction that invalidates T's operations is allowed to commit at that server. That is, until T has finished validation, any transaction that conflicts with T's operations will be aborted (or delayed depending on the TM's choice). Thus, at T's commit time, any timestamp value in the range [*curtime*, ∞] can be chosen for T.ts. The TM achieves the effect of assigning a timestamp range to transaction T by creating two entries for T in the VQ — one with *curtime* for T.ts and the other entry for which T.ts is infinity. If T is serializable at both these points, it is serializable at any point in between[2].

By early-preparing T in this manner, the TM prevents transactions conflicting with T from committing at the server. This strategy could be used to implement locking. Acquiring a read (write) lock for object x on behalf of transaction T corresponds to early-preparing T at the site such that $x \in$ T.ROS (T.MOS). If T fails the early-validation process, the TM will delay T till the lock request can be satisfied. As part of the lock grant message, the server can also inform the frontend about the new lower bound on T.ts (if the TM was not able to serialize T on the *curtime* value suggested by the frontend). Object fetches are not affected by this mechanism; transactions that do not request locks are allowed to fetch "locked" objects.

Out of the three approaches discussed, early send seems to be the most useful strategy since it amortizes the commit delay over a transaction's lifetime by sending the commit information during the transaction's execution phase. Early validation just consumes more network bandwidth and increases the load on the frontend/servers. If a locking mechanism is to be provided in Thor, the early prepare mechanism could be used to implement locks. Otherwise, like early validation, it does not offer any significant advantages.

4.3 Asynchronous commit

The optimizations discussed so far decrease the foreground delay of the commit protocol. However, even if all the optimizations suggested in the previous sections were implemented, a transaction commit call will be more expensive than a normal operation call due to the messages and log flushes involved. To avoid performance problems, a client-application must be designed with the knowledge that the application will be delayed at commit time. In this section, we propose a mechanism that gives the application better flexibility and control over commit time delays. To obtain benefits from this facility, the application programmer needs to modify his code structure slightly.

In this strategy, the application sends a message to the frontend for committing its current transaction T. On receiving this message, the frontend records the commit request and returns control to the application. The application proceeds with the execution of the next transaction while T's commit protocol is being carried out by the frontend. Since the

[2] In general, if T passes validation for timestamp values ts_1 and ts_2, it can successfully validate with any timestamp value in the range [ts_1, ts_2]; the VQ-check guarantees this property.

application does not wait for T to be committed/aborted, it observes T's commit delay to be the same as the delay observed for any normal operation. This strategy of committing transactions without waiting for their result will be termed *asynchronous commit.* The original commit approach in which the application waits for the commit result will be referred to as *synchronous commit.* In the asynchronous commit scheme, the application needs some way of determining whether T committed or not; the application interface has to be modified to provide an operation that the application can call to determine T's commit result (see Section 4.3.2).

The Thor model assumes that aborts are rare; with this assumption, it is wasteful for the application to wait on commit results. The asynchronous commit mechanism provides the flexibility to avoid such waits and is in consonance with the basic Thor philosophy of optimizing the normal case. Note that this mechanism does not reduce the work done by the frontend; it just decreases commit time delay observed by an application. If an application is overly aggressive in committing transactions, its performance *will* be degraded due to limitations of processor speed, network/disk bandwidth, etc.

Any application that wants to overlap the validation phase of a transaction with the execution phase of a later transaction can benefit by the asynchronous commit mechanism. This facility is especially useful for applications that have *little or no sharing*, *very few conflicts* and *need Thor mainly for persistence*; using asynchronous commit, they incur very small overhead due to the concurrency control mechanism. Essentially, this commit facility provides a cheap way of writing objects atomically, making Thor sufficiently lightweight to be used for many applications, *e.g.,* writing files, single user applications, etc. Some other applications that can use this facility are:

- A real time display application that is reading a set of objects and displaying objects on the screen can take advantageous of this facility. The display program reads some objects, performs an asynchronous commit and copies the objects into a buffer. While the commit is proceeding, it can start the next transaction and process data for the subsequent set of objects to be displayed. Whenever it knows that the transaction has committed, it can display the objects from the buffer.

- In a CAD application for designing an integrated chip, each user is assigned a different part of a chip. A user operates on a low-conflict database, essentially a private database that is available to other users but rarely used by them. Hence, a transaction committed by such a user usually passes validation. A user may prefer to have the commit call return quickly and be notified of a failure later rather than being delayed every time he commits a few changes.

- Small transactions (not many objects have been read) can be committed frequently without excessive degradation in performance. For example, a shared editor program that treats each word as an object can commit the changes at the completion of a line. Without the asynchronous commit facility, the editor would have poor performance. As stated earlier, this mechanism does not reduce the total work done by the frontend/servers and an editor is unlikely to perform well if it commits too frequently.

4.3.1 Asynchronous Commit Issues

Asynchronous commit raises some interesting issues regarding multiple commit calls received by a frontend. Consider a scenario in which an application commits transaction T_1 followed by T_2, both in the asynchronous mode. Suppose that the first phase of T_1 is not finished when the frontend receives the commit call for T_2. This situation is unlikely for interactive applications, but is possible for non-interactive applications with a high transaction commit rate. There are two choices for the frontend:

1. Wait until T_1 commits and then start the first phase of T_2, *i.e.,* there is at most one pending commit at a frontend. This scheme is called *single pending commit* or SPC. When the asynchronous commit call for transaction T_2 returns, the application knows about the outcome of all transactions that committed before T_2. This approach is well-suited to interactive applications where the duration of a transaction is larger than the time taken to complete the first phase of the commit protocol. Furthermore, it is easier to write an application program with at most one unknown commit result (see Section 4.3.2).

2. Allow T_2 to be committed in parallel with T_1. When the application commits a transaction, it no longer has the guarantee of knowing the commit results of previously committed transactions, *i.e.,* there are can be *multiple pending commits* at a frontend. This approach makes it more difficult to write application programs, but it provides a facility for committing small non-interactive transactions frequently.

There are some problems in supporting the second approach. Suppose that T_1 has modified object x_v to x_{v+1}. The application commits T_1 in asynchronous mode and starts processing T_2. T_2 reads object x (the frontend returns the object value that has been written by T_1, *i.e.,* x_{v+1}) and is also committed in asynchronous mode. The frontend does not know T_1's result but it starts T_2's commit process in parallel. T_1 fails validation and aborts. Meanwhile, a transaction S from another application commits and updates x_v to x_{v+1}. T_2 reaches the relevant server and passes validation although it should have been aborted. Essentially, x's version number did not have sufficient information about the transaction that had installed x_{v+1}. If the version number field is implemented using timestamps[3], the server can detect that T_2 had not read x_{v+1} from S and abort T_2.

A problem not alleviated by using timestamps is that of cascading aborts, *i.e.,* transactions whose results are not known may be dependent on each other and aborting one of them may abort a later transaction and so on. The frontend can alleviate this situation by delaying an asynchronous transaction commit if it depends on an earlier transaction whose result is not known. Suppose an application has committed transactions T_1 and T_2 (in that order) in asynchronous mode. T_2's commit is allowed to proceed in parallel with T_1's commit if T_2 does not depends on T_1, *i.e.,* if T_2 has not read any object written by T_1:

$$T_1.\textsc{mos} \cap T_2.\textsc{ros} = \phi \qquad \textit{Condition 1}$$

[3] Recall that each transaction is assigned a globally unique timestamp that can be used to identify a transaction.

This is called the *independent multiple pending commit* or IMPC scheme. If T_2 is dependent on T_1, the application call will block till T_1's outcome is known.

However, the possibility of cascading aborts still exists. Suppose T_2 has modified an object y that T_1 has read. If messages are reordered and T_2 installs y's new version before T_1 reaches the server, T_1 fails validation and is aborted. Although the likelihood of aborts due to reordering of messages is very low, they can be avoided by allowing T_2's commit to proceed in parallel with T_1's commit if both condition 1 and the following condition are satisfied (this is called the *non-conflicting multiple pending commit* or NMPC scheme):

$$T_1.\text{ROS} \cap T_2.\text{MOS} = \phi \qquad\qquad \textit{Condition 2}$$

The NMPC approach has the advantage that it supports stronger semantics, *i.e.,* an application is given the guarantee that only the results of non-conflicting transactions are not known. Thus, an application's call for committing its current transaction T blocks at the frontend until all transactions conflicting with T have committed/aborted.

Multiple pending commits seem very attractive, however it is unlikely that consecutive transactions committed by an application are independent of each other. Furthermore, given the assumption that the commit time delay is much less than the execution time of a transaction (for interactive applications), the multiple pending facility will be of little use. Thus, for most cases, the SPC scheme will suffice. For the rest of this section, we will not consider the MPC schemes.

4.3.2 Application Interface

As discussed in Section 4.3.1, different schemes that support varying levels of control and flexibility can be implemented. Let us see how the application interface is modified to support the SPC scheme. A *transaction object* type is provided by the frontend to support the asynchronous commit mechanism. When an application asynchronously commits a transaction, the frontend returns a handle to a transaction object. The application can call various methods on this object, *e.g.,* determine the status of transaction. Note that transaction objects are not essential for supporting the SPC scheme; they just make the application code structure simpler. Thus, two features are added to the application interface — an *async-commit* call and the notion of a transaction object:

1. **Async-commit()** — This procedure commits the current transaction T in asynchronous mode and returns a handle to a transaction object corresponding to T.

2. **Transaction Object:** Some of the important methods of the transaction object type are:

   (a) **Status()** — The application can inquire about the status of a transaction using this call. For example, if the application wants to ask about the status of transaction T whose transaction object is *tr*, it simply calls *tr.status()*. The frontend can respond with one of three: *committed*, *aborted* or *not-known-yet*.

   (b) **Block-until-commit()** — This call blocks until the result of the relevant transaction is known and returns the outcome.

(a) Synchronous commit

do
    ...
    % The code for transaction T_1.
until commit()   *% The application waits for* T_1 *to commit.*
  do ...
    % The code for transaction T_2.
until commit()   *% The application waits for* T_2 *to commit.*

(b) Asynchronous commit

result_1 := false; result_2 := false
do
    if (not result_1) then
        ...
        % The code for transaction T_1.
        tr_1 := async-commit()
        endif
    ...
    % The code for transaction T_2.
    % T_1*'s validation phase being overlapped with* T_2*'s execution phase.*

    result_1 := tr_1.block-until-commit()
    % Usually no waiting; T_1*'s result is known.*
    if (result_1) then
        result_2 := commit()   *% The application waits for* T_2 *to commit.*
        else
            result_2 := abort()   *%* $\textit{result}_2$ *is set to false.*
        endif
until (result_1 and result_2)

Figure 4-4: Code restructuring for asynchronous commit

Let us see how an application might use the asynchronous commit facility. Suppose that it wants to commit transactions T_1 and T_2 with two requirements. Firstly, the work done by T_1 and T_2 should be done exactly once. Secondly, it wants order to be maintained, *i.e.,* T_2 must follow T_1. Figure 4-4 shows the code structure for committing these transactions in the synchronous and asynchronous modes. For the sake of simplicity, assume that the result of a transaction is true or false (for commit/abort). In the synchronous case, the application keeps executing a transaction until it commits. Furthermore, it starts T_2 after T_1 commits, so the order property is guaranteed.

In the asynchronous commit case (Figure 4-4b), the application asynchronously commits T_1 and starts T_2. At the end of T_2, it checks for T_1's outcome. If T_1 has committed, the application commits T_2 else it aborts T_2 and restarts T_1. Clearly, T_1 and T_2 are executed exactly once. Furthermore, the order property is also maintained since T_2 is committed iff T_1 has committed. Thus, the application's requirements can be met while using asynchronous commit if the code structure is altered slightly. The difference in the synchronous and asynchronous case is that the latter has a better performance since there is no waiting for T_1's commit. Note that T_1's result will usually be known when the block-until-commit call is made, so no waiting needs to done at that point. An issue of concern is that T_2's work is wasted in the asynchronous case if T_1 aborts. To reduce the amount of wasted work, the application can check for T_1's result before T_2 is finished and continue with T_2's computation iff T_1 has succeeded in committing. The code shown in Figure 4-4b has been structured to demonstrate the normal case where T_1 would have succeeding in committing; T_1's result is checked after T_2's work has been completed.

The previous example demonstrated how asynchronous commit can be used to commit two transactions with the same semantics as the synchronous case. In general, the code structure will depend on the relationship that the application wants to maintain between T_1 and T_2. Essentially, what it wants to do with T_2 if transaction T_1 aborts will determine the control flow of the code.

In conclusion, asynchronous commit with single pending commit is a useful strategy to support since neither the implementation nor the application interface are complex/inefficient. Asynchronous commit provides a limited form of multi-threading without making the code-structure excessively complicated; providing support for multi-threading would require a much more sophisticated application interface. Furthermore, asynchronous commit is also in consonance with the basic Thor philosophy of optimizing the normal case (very few aborts); it provides a facility in which the application can tradeoff code simplicity for better performance.

Chapter 5

Object Migration

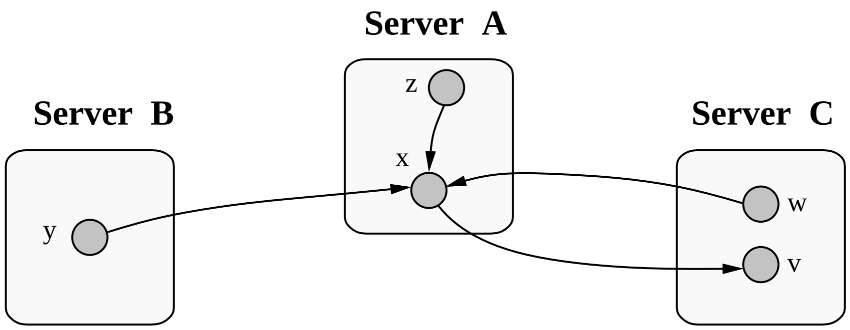
The discussion in the previous chapters assumed that objects remain at the server where they were initially installed. This chapter explores the idea of moving objects among servers and discusses various issues related to object mobility. The migration facility designed for Thor is essentially available to applications for performance enhancement purposes.

Any system that is intended to be used over a long period of time requires the ability to migrate objects for reconfiguring object placement in an application-specific manner. Some scenarios where the object migration facility will be useful are:

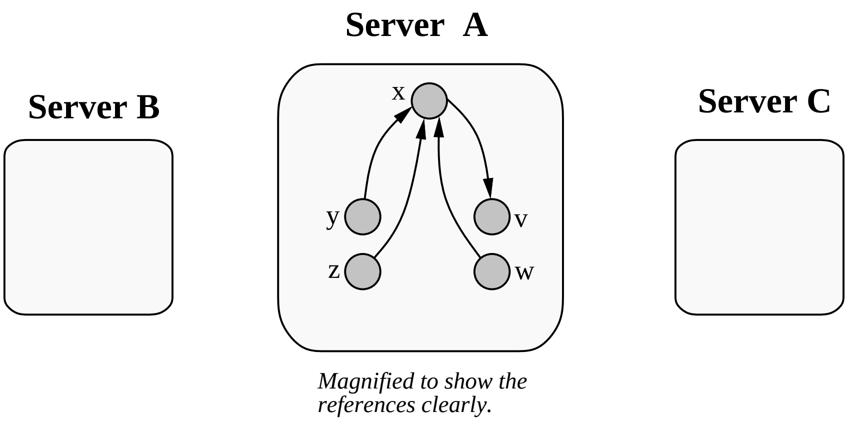
- An application may migrate its objects to a single server to improve its performance. Clustering an application's objects at a single server has the advantage that a 2-phase commit protocol does not have to be executed to commit a transaction.

- The migration facility may also be used for reducing the number of inter-server or external references among objects. Fewer external references are beneficial for the distributed garbage algorithm [ref] and may cause prefetching to be more effective. As discussed in Chapter 2, prefetching may be done by following the references of a fetched object, but only when they refer to objects at the same server. More effective prefetching may be achieved if there are fewer external references because a significant fraction of an application's working set would be brought to the frontend cache as prefetched objects instead of the frontend sending explicit messages and waiting for them. When objects are created, the application or the system tries to place objects such that inter-server references are minimized. But the initial object placement may become less effective and more inter-server references may be introduced as the system is used. Object migration can be used to reduce the number of external references by reconfiguring the placement of objects, *e.g.,* an object can be moved to a server where most references to it reside or vice-versa.

  Figure 5-1(a) shows a scenario where there is a significant number of remote references between various objects. Migrating objects v, w and y to server A results in the situation shown in Figure 5-1(b), where all the inter-server references have been converted to local references.



(a) ***Before object migration***



(b) ***After object migration***

Figure 5-1: Reducing the number of external references using object migration

- An application may migrate objects due to the physical movement of the corresponding (physical) entities. For example, a car company may have its factory at site B and part of its inventory at site A. When the parts are shipped from site A to B, the objects corresponding to the moved parts may also be migrated to a server at site B.

- An application may want to use a group of objects for a short period of time. If it is the only client using these objects, it can migrate these objects to a server that is closer to its site; accessing objects from a server closer to an application site may improve the application's performance.

- An application may move rarely-used objects to another server to avoid cluttering up a particular server. For example, in a company database, old records can be moved to another server for archival purposes. This will ensure that the commonly used server stores the recent objects only.

- If a machine has to be shutdown for some time, important objects can be migrated to other servers. Since Thor provides high availability, object migration is not required for this purpose. But if a server has to be removed from service, object migration can be used to move all its objects to other servers.

- If a server becomes heavily loaded with a large number of objects, it will start affecting the performance of the system. Objects can be moved to other or new servers, *i.e.,* the load of this server can be decreased by spreading it across the system. Note that load balancing and clustering all objects at one server are opposing requirements. Object migration provides a mechanism for an application to reconfigure object placement to suit its requirements of load balancing and clustering.
- The system may expand or shrink by removal or addition of new servers during the system's lifetime and migration may be required to reconfigure the object placement.
- An application can move the objects to a server if that machine has some special software or hardware characteristics, *e.g.,* a faster processor or bigger cache.

Note that an application can migrate an object x by copying x from its original site A to its destination site B. But it has to change all the references to object x to refer to the new location; this process is complex and inefficient. The object migration facility provides a transparent mechanism that removes the burden of such activities from the application and also has better performance.

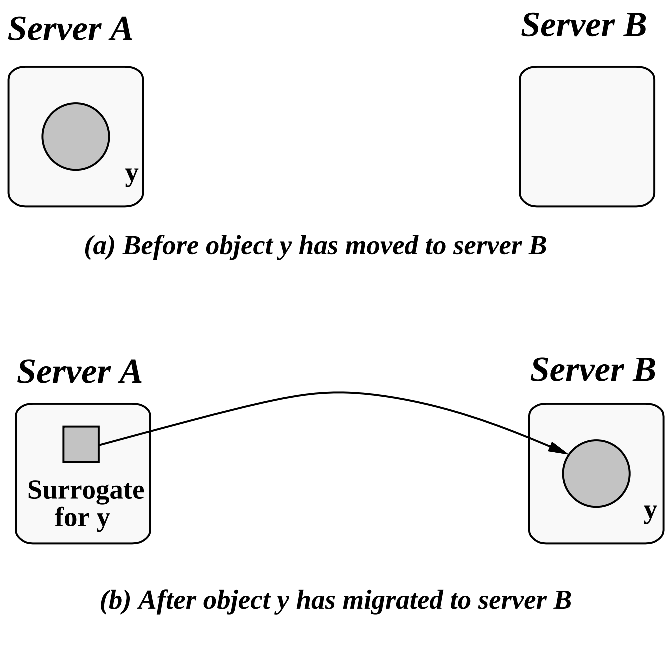


Figure 5-2: Use of surrogates for moving objects

In our design, when an object y is moved from A to B, it is replaced at server A by a *surrogate* or forwarding pointer that contains the xref of y at server B (see Figure 5-2). Any operation that tries to fetch y from A is automatically forwarded to server B. Thus, all the old references to x are still valid, although an extra level of indirection has been added. These indirect links are snapped later and when no more references to the surrogate exist in the system, the surrogate is garbage collected.

An important assumption made in our design is that objects are moved rarely. We also assume that information about a moved object x is usually propagated to frontends and servers that have references to x before x moves again. We take advantage of these characteristics and ensure that the common case is not penalized, *i.e.,* when there are few or no object migrations.

5.1 Semantic Issues

In this section, we discuss the interface issues about object movement, *i.e.,* an application's view of moving and locating objects. The following application interface issues are discussed:

- *The primitives available for moving and locating objects.* An application needs some way of indicating how and where it wants the objects to be placed.
- *Interpretation of an object move.* Are object locates and moves independent of reads/writes or are they related to these operations?
- *The atomicity guarantee with respect to migrating objects.* Are the relevant objects moved iff the transaction commits or are the move primitives mere hints to the system?
- *The relationship between the location and movement primitives.*

In our design, semantics of object migration have been chosen such that it is easy for programmers to reason about their programs. Intuitively speaking, object mobility should be orthogonal to the object state or type. Thus, migration has been designed to be independent not only of an object's type but also of read/write operations. Furthermore, transaction semantics of atomicity and serializability are provided with respect to migration resulting in uniform semantics for all user operations. These issues are discussed in more detail in the next few subsections.

The rest of the chapter assumes the existence of a distinguished and immobile *node-object* for each server. The move/locate primitives can refer to a particular server using its node-object. Thor has such an object as a fundamental part of its design — the server's root directory.

5.1.1 Primitives for Locating and Moving Objects

In this section, we discuss the primitives to locate and move objects. These primitives can be executed as part of a transaction along with reads and writes.

An application can locate objects and based on their location, it may decide to move some of them to a certain server. For example, to minimize object movement, the application can determine the location of objects x_1, x_2, $\dots x_n$ and then move all the objects to the server where most of the x_i's reside. To determine an object's location, the application can use the *locate-object* primitive:

$$\textit{locate-object} = \textbf{proc} \text{ (Object z) } \textbf{returns} \text{ (Node-object)}$$

The locate-object primitive returns the node-object corresponding to the server (say A) where z was located in the recent past. It only guarantees that z resided at A recently because some other application may have moved the object to another server. But most likely z is still located at server A since object moves are assumed to be rare and we propagate the migration information quickly to the relevant frontends and servers. Note that the locate-object call is similar to the read operation. In the former case, an application may have read an old location; similarly, in the latter case, the application may have observed an old value.

The frontend implements the locate-object primitive by simply determining z's server using z's xref[1]; this call does not require any communication between the frontend and the server. As stated earlier, z need not be located at A when the call returns.

To move an object x, the application uses the *move-to-server* primitive:

$$\textit{move-to-server} = \textbf{proc} \text{ (Object x, Node-object N)}$$

This procedure asks the frontend to move object x to the server whose node-object is N. The frontend records this request and returns the control to the application. The frontend groups all move requests and performs them at transaction commit time. By performing the object moves as part of the commit protocol, the system is able to reduce the application delay and reduce network bandwidth requirements. Furthermore, strong guarantees of atomicity and serializability can be provided only if object moves are performed at commit time.

5.1.2 Object Location with Respect to Object State

The location of an object can be viewed as part of the object state. This implies that locating an object is equivalent to reading it and moving an object modifies its value. Such a scheme would force atomicity and serializability to be provided with respect to object migration because these semantics are guaranteed for reads and writes. However, this approach has some disadvantages. Firstly, treating a move like an update unnecessarily increases the percentage of writes on the database. This can cause *false* conflicts between moves, locates, reads, and writes resulting in unnecessary aborts. Secondly, migration becomes type dependent. Objects of immutable types cannot be modified, so it is not possible to treat an object move like an update in a uniform manner; either migration of immutable objects has to be disallowed or different semantics must be supported for immutable objects. Thus, treating an object's location to be part of its state causes unnecessary aborts and is non-intuitive.

To achieve independence of an object's location from its state, we partition an object's state into two parts — the *value-state* or the state required for reads/writes and the *location-state* or the state required for locate/moves. Note that the value-state actually corresponds to some attributes present in the object whereas the location-state is just a logical concept

[1]If the application has a handle to an object z, either z or its frontend-surrogate is present at the frontend and the server number can be extracted from the xref.

and does not consume any space in the object; the server where an object resides represents its location-state. The locate-object primitive *reads* the location-state and move-to-server operation *modifies* it. The former is a shared operation and the latter is an exclusive operation with respect to the location-state; they are independent of reads/writes on the value-state.

Apart from the advantages discussed above, partitioning the object state in this manner gives us the flexibility of not providing transaction semantics for object migration (although as discussed in the next few subsections, we do guarantee transaction semantics with respect to migration primitives also). Furthermore, this approach does not require a located or moved object to be present in the frontend cache; the object's xref is sufficient to provide the necessary information[2]. However, as discussed in Section 5.2.1, making object migration orthogonal to reads/writes does add some complexity to the commit protocol.

5.1.3 Atomicity of Object Moves

Suppose that an application asks its frontend to move x_1, x_2, ..., x_n to server A. If there is no guarantee that either all or none of the x_i's will be moved to A, the system may migrate some of the objects and leave others at their source servers. Such *partial migration* of objects can increase the number of inter-server references among the migrating objects and degrade the system's performance. To prevent such possibilities, the system must ensure all-or-nothing semantics for object movement, *i.e.,* either all objects are moved to the desired destinations or none of them move. Our migration mechanism guarantees atomic movement of objects; all relevant objects are moved if and only if the transaction commits. To support atomicity with respect to object migration, the commit protocol has to be changed; each destination server must reserve sufficient space in the validation phase to ensure that these objects can be migrated in the installation phase.

5.1.4 Relationship Between Locates and Moves

When an object is located by an application, its frontend simply returns the server number from the object's xref. However, the object may have moved by the time the transaction commits, *i.e.,* the object's location-state may have changed. The system can either ignore the fact that the transaction has read an old value of the location-state or treat the locate as invalid and abort the transaction. The former approach has the advantage that the server does not have to check if a located object has moved or not. But it offers weaker semantics compared to the latter approach in which locates and moves are serializable with respect to each other. In the latter approach, a transaction T commits successfully with timestamp T.ts only if all its locates and moves can occur at time T.ts in an equivalent serial schedule. Therefore, like reads and writes, object locates and moves must be validated at the end of a transaction. This approach makes it easier for users to reason about their programs since transaction semantics are being supported for all operations on the value-state *and*

[2]If an object's location is part of the the object state, the object has to present in the frontend cache so that its version number can be recorded.

the location-state.

Let us consider an example to understand the transaction semantics offered with respect to object migration. Suppose two applications want to co-locate objects x and y that are currently located at servers A and B respectively. The first application locates x's server to be A and moves y to that server. The second application determines y's server to be B and asks x to be moved to B. Figure 5-3 illustrates transactions T and U executed by the two applications respectively. T reaches the two servers before U and is able to move x to A. U aborts because it is not serializable with T and y is not moved to server A. If locates were not validated, y would have been moved to server A and x moved to server B. Therefore, we have chosen the scheme in which locates are also validated at the end of a transaction; if an object located by transaction T has moved to another server, T is aborted.

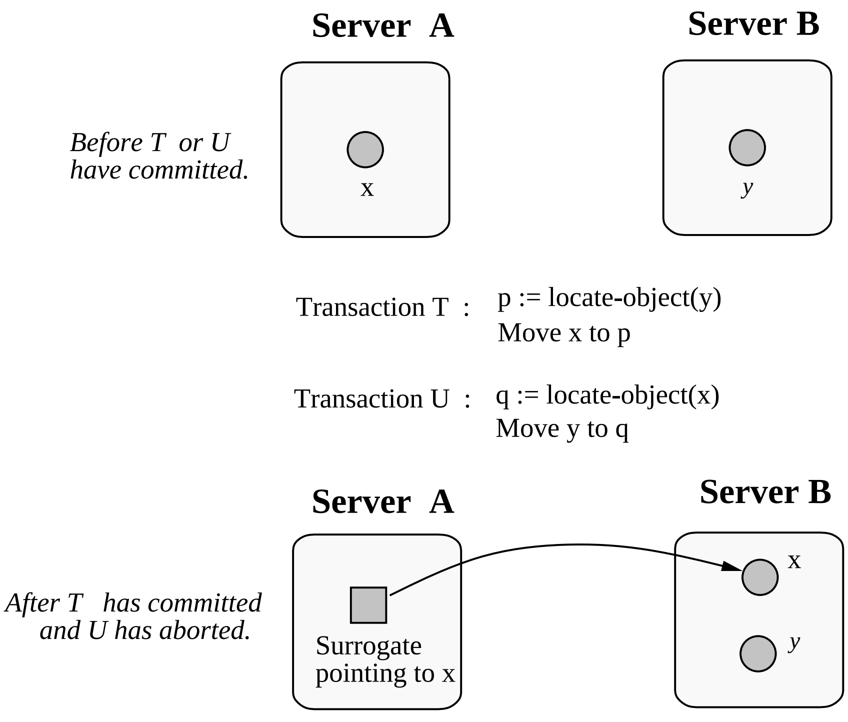


Figure 5-3: Serializability of transactions with respect to move/locate primitives

5.2 Migration Mechanics

In this section, we discuss how atomicity and serializability with respect to migration can be provided. Since the locate-object operation reads the location-state and the move-to-server operation modifies it, a validation scheme for the location-state can be designed that is similar to the one designed for the value-state. Section 5.2.1 discusses the changes made to the validation algorithm for accommodating migration and Section 5.2.2 presents the installation phase issues.

5.2.1 Validation phase

When an application commits its current transaction, T, the validation algorithm must not only check the validity of reads/writes but also of moves and locates. The frontend chooses a coordinator for T and sends it the following sets:

1. **Read Object Set or** ROS — The set of objects read by T.
2. **Modified Object Set or** MOS — The set of objects modified by T.
3. **New Object Set or** NOS — The set of objects that are being made persistent for the first time; NOS objects are installed iff T commits.
4. **Locate Object Set or** LOS — The objects that have been located by T, *i.e.,* the application has used the locate-object primitive for these objects. This set just contains the xrefs of the located objects.
5. **Migrating Object Set or** MIOS — The set of objects that the application has asked the frontend to move. Each set element is a tuple of the form <object, xref, size, server>, *i.e.,* the object, its xref and its destination server. The size field contains the current size of the object known at the frontend; if the object was not cached in the frontend cache, the value of the size field is zero. During T's execution, if the application has asked object x to be moved to two different servers, then only the last call is considered. That is, if a move-to-server(x, A) is followed by move-to-server(x, B), then the destination server for x in MIOS is B.

The frontend sends this information to the coordinator and waits for the reply. The coordinator chooses a timestamp for T and sends it along with this information to the participants. T's participants are the servers from which objects have been read (ROS), modified (MOS), created (NOS), located (LOS) and moved to (MIOS) by T. In case an object is being moved from site A to site B, the coordinator sends prepare messages to both A and B. As in the MOS case, we assume that *blind moves* are not allowed; when an application executes a move-to server operation for a particular object x, the system automatically executes a locate-object operation for x. Blind moves, like blind writes, are not permitted since they complicate the algorithm (this issue is discussed in Section 5.3). Thus, we can assume that the MIOS is a subset of the LOS; similar to the case of object writes where MOS $\subseteq$ ROS.

On receiving the prepare message, each participant performs the watermark, version and VQ tests for the value-state. Along with these tests, it also validates T's moves and locates using the following checks:

1. **Location Check** — This test is used to validate the locates executed as part of the incoming transaction. It is analogous to the version check for the value-state. As described in Chapter 3, the version number of an object x truncates x's modification history. Similarly, x's server number acts like the version number for the location-state[3]. As the version number is incremented on every updated, an object's server

[3]Actually, the server number represents x's location-state *and* also functions like a version number for this state.

number is also "modified" whenever the object moves. To validate a transaction's reads against committed transactions, the TM performs the version check to verify that T has read the latest version of each object. Similarly, a check is required for all the LOS objects to ensure that T has read the latest value of the location-state for each object x in the LOS. In other words, a participant server B has to ensure that none of the LOS objects have moved from B to some other server. The server simply performs this check by verifying that x is not a remote surrogate. This test is called the *Location Check*.

Consider the scenario in which an object x moves from site A to B and back to A. Suppose that x has been located using its old xref at site A by transaction T. If x has been moved back to its original xref at A, T will pass the location check. However, if a new xref has been allocated for x, T will abort; this is so because the old xref at A will be a remote surrogate that points to x's place at B.

2. **Watermark Checks** — Similar to the watermark checks for the value-state, these tests are required because there is no timestamp information being kept about the transaction that last located or moved a particular object. Thus, like the read and write watermarks, the transaction manager needs to maintain a *locate-watermark* (X_l) and *move-watermark* (X_m). The locate-watermark for server A denotes the timestamp of the latest transaction that has located an object at A. The move-watermark denotes the timestamp of the latest transaction that has moved an object from or to server A. As in the case of the value-state, these watermarks are updated after a transaction has committed at that site and its entry has been removed from the VQ.

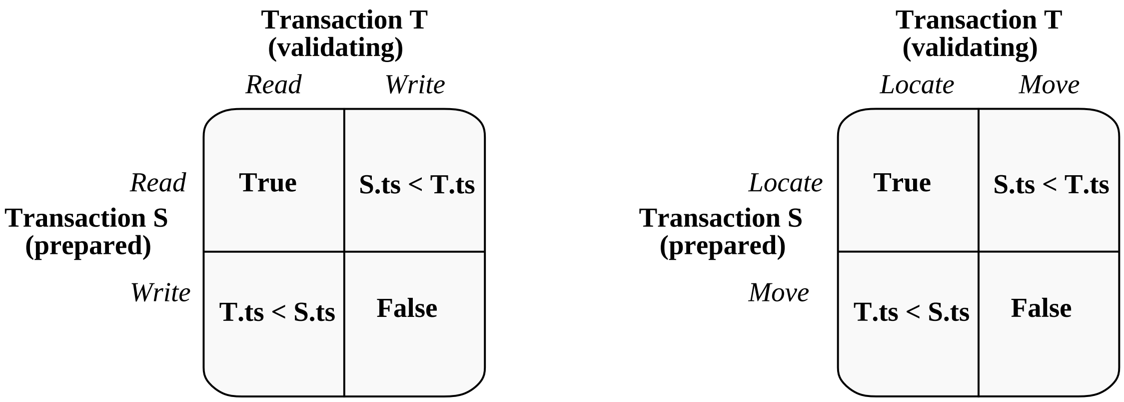


Figure 5-4: Validation Queue check

3. VQ **Check** — The VQ check for the location-state verifies that none of the moves/ locates of the incoming transaction T have been invalidated by a prepared transaction. T passes the VQ check only if it satisfies the following conditions for each prepared transaction S:

   (a) If x ∈ T.LOS and x ∈ S.MIOS then T.ts < S.ts. <u>VQ-*rule*$_a$</u>

   (b) If x ∈ T.MIOS and x ∈ S.LOS then S.ts < T.ts. <u>VQ-*rule*$_b$</u>

These rules bear a strong resemblance to the VQ-check for the ROS/MOS case. VQ-rule$_a$ prevents T from validating if S is moving x and has a lower timestamp than T. T is aborted because at time T.ts (in an equivalent serial schedule), x would not be present at server A. Similarly, VQ-rule$_b$ prevents T from validating if one of its move primitives has been invalidated by a locate-object executed by S. Figure 5-4 gives a graphical representation of the VQ-rules. If S and T have operated (read/written/located/moved) on an object x, the relevant table entry is used to validate T. T passes the VQ check if for each x, the condition specified in the corresponding table entry is true. For example, if S has located an object and T is moving it, T passes validation only if S.ts < T.ts. Similarly, if S and T are both moving objects, T fails validation (blind moves are not allowed).

Interactions Between the Location and Value States

There are some interactions between the location-state and the value-state that may result in extra messages being sent in the validation phase of the commit protocol or cause delays in object fetches. Consider the following scenarios (T is a validating transaction, S is prepared at site A and object x is currently located at A):

Scenario 1:

Suppose object x is being moved to from server A to server B by transaction T. The coordinator (server C) *predicts* x's size and asks space to be reserved at B. The coordinator can predict x's size with the help of the MIOS information sent by the frontend. The destination server B must ensure that sufficient space is available for x to be moved to B. But two extra foreground messages may have to be sent for reserving space at the destination server. Suppose transaction S is modifying x and increasing its size. The space reserved at server B on behalf of T corresponds to the current value of x. Since S is increasing x's size, server B must have sufficient space to store x's potential version (the version S is going to install). This has to be done to take care of the fact that S may commit or abort. Server A sends a *space-reserve* message to server B to reserve space for object x. Server A can either wait for the acknowledgement from B and then respond to the coordinator or it can ask the latter to wait for B's response. The second scheme is better because it avoids a foreground message delay. Note that B may or may not be able to combine the *space-granted* message with its vote message. Scenario 1 has been demonstrated in Figure 5-5 (all messages are sent on T's behalf). In the figure, we have shown the case in which B sends separate vote and space-granted messages to the coordinator.

There are situations similar to the above scenario that may require extra foreground messages. For example, suppose T is modifying x and S is moving it from A to B. If the new version of x that T is trying to install has a size greater than the space reserved for it on B, a space-reserve message has to be sent to B. Note that these problems would have been avoided if moves were treated like writes and locates like reads. Instead of sending space-reserve messages to the relevant servers, the system would aborted the validating transaction because moves and writes would have conflicted.

Scenario 2:

Suppose T has read object x from site A but x has moved to B. To validate T's read, the

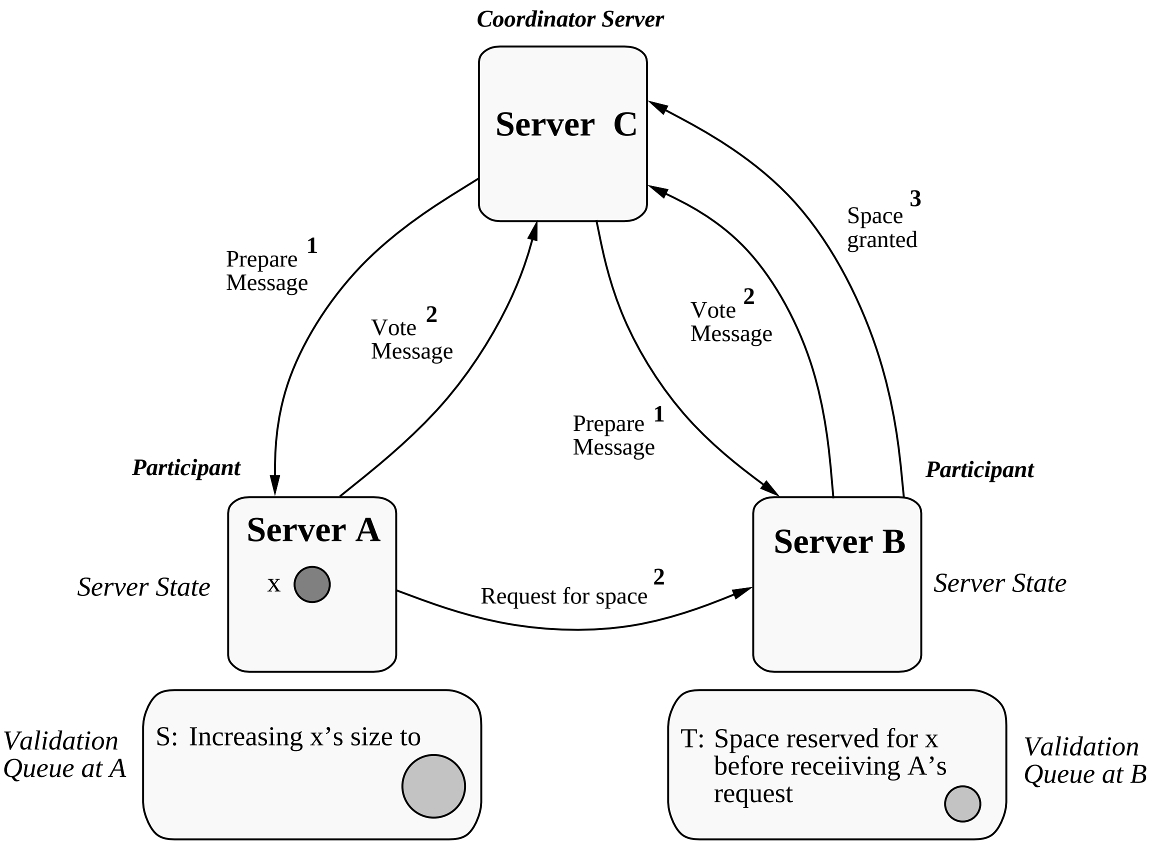


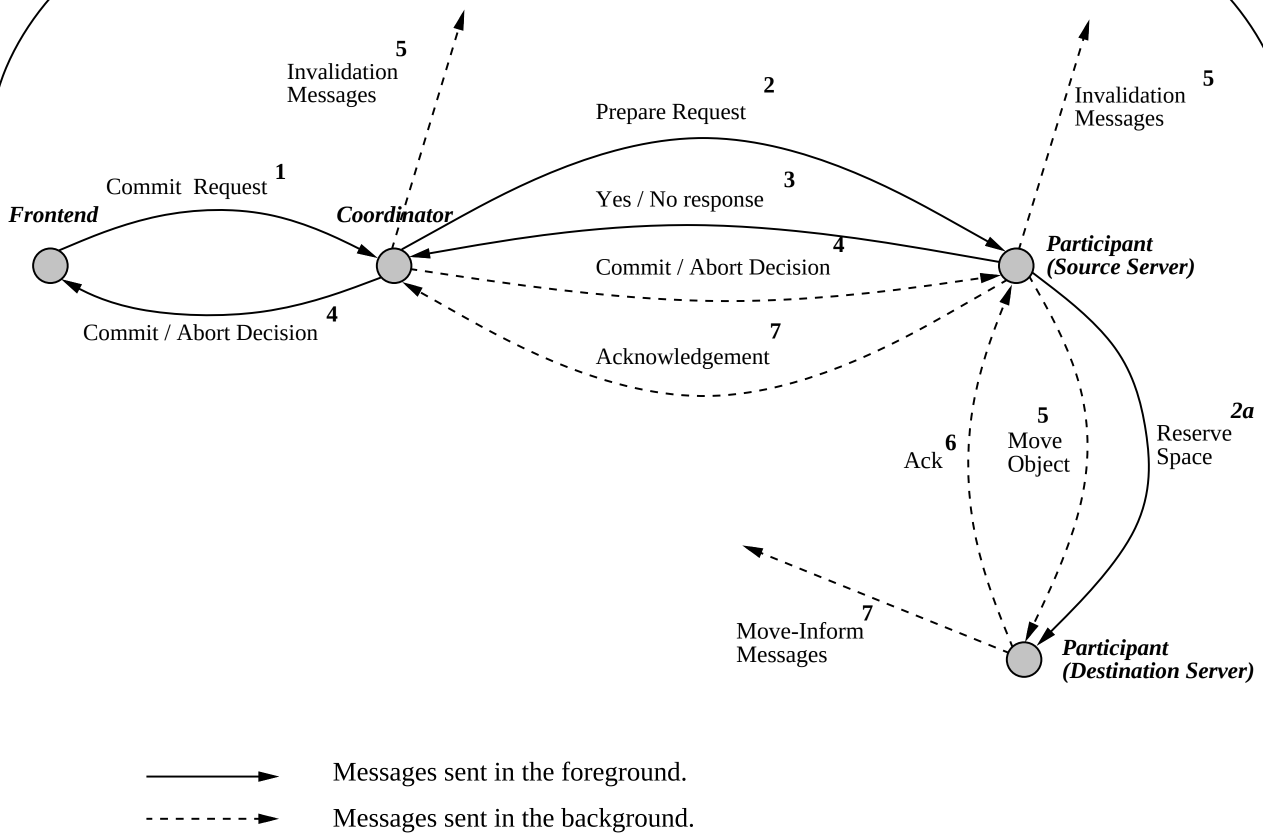
Figure 5-5: Reserving space for migrating objects

TM at site A must forward the prepare message to B. If x has moved from B to another site, this message has to be forwarded to that server. This forwarding process continues till the x's current location is found; T's read is validated at that server. This is a rare situation because object moves are rare and long surrogate chains are unlikely to exist. Furthermore, since servers inform frontends about the recent moves (see Section 5.2.2), frontends usually know the current locations of objects.

A similar situation can arise in the installation phase. Suppose transaction T is modifying object x at server A and x has migrated to server B. When T receives its commit message from the coordinator, this message must be forwarded to B so that object x can be modified at B. As in the above scenario, the commit message may have to be forwarded until x's current location is found.

Scenario 3:

Suppose that transaction T is moving object x from server A to B. It passes all the validation and space allocation checks at both servers. The destination server B allocates a new xref for x and returns this information to the coordinator. T commits and the coordinator sends these xrefs along with the NOS xrefs to the participants and the frontend. Now the frontend



Notes

1. The coordinator also sends a prepare message to the destination server asking it to reserve space for the migrating objects. For the sake of clarity, the messages between the coordinator and the desination server are not shown.

2. The message 2a is sent only if space reserved at the destination server is less than the space taken up by the object at the source server.

3. Coordinator will wait for ok from destination server before ending phase 1 if the source server has sent a space-reserve message.

Figure 5-6: The two phase commit protocol modified for object migration asks x to be fetched from its new server (*i.e.,* server B). If x has not been migrated to server B when the fetch arrives at B, server B can either delay the fetch till x's migration has been completed or it can declare the fetch as invalid. Note that this situation is similar to the case where a frontend may fetch an object before it has been installed at the server.

5.2.2 Installation Phase

On receiving the commit decision from the coordinator, each participant server initiates a move protocol to migrate the relevant MIOS objects to the destination servers. Suppose that x is stored on server A and a committing transaction T is moving x to B. Server A has to ensure that the latest value of x is installed at B. Furthermore, A must also send the relevant validation information to server B. In particular, the read/write watermark and VQ information must be sent to B.

Moving x from A to B is similar to installing the update of an object. In the latter case, the object is locked with respect to validation and fetches, the update installed and the lock released. To move an object x, the TM at A keeps x locked until x has been moved to server B. This lock prevents validation and installation from occurring while the move is going on. Object fetch need not be blocked; it is delayed only when the object is converted to a surrogate — a short term synchronization lock.

To perform the move, server A locks x and sends its latest value to B. Along with the VQ information and the read/write watermarks, A also sends the frontend-table and inlist information relevant to x. It also merges the VQ information with its VQ and updates its watermarks to be the maximum of the original values, incoming values and T.ts. Note that server A does not have to send its locate/move watermarks to server B. Information about prepared transactions that have located x also need not be sent to B. This is so because T.ts is an upper bound on the timestamp of any prepared transaction that could have located x, *i.e.,* T.ts captures the locate/move watermark and the relevant prepared transaction information.

After completing the installation process for x, B sends an acknowledgement message to A. On receiving this message, server A converts x into a surrogate and forwards all transactions waiting on x to server B. Server A also updates its locate/move watermarks to be the maximum of the old values and T.ts. Meanwhile, server B sends *move-inform* messages to frontends that have cached x. These messages are similar to invalidation messages sent for modified objects. Server B can also send these move-inform messages to servers that have remote pointers to x; it can determine these servers using the inlist information. It is not necessary to send this information in separate messages; it can be piggybacked on other messages. When the servers receive this information, they can change their pointers to x's new location. This prevents long surrogate chains from being formed.

If transaction T is moving more than one object from server A to server B, T can group all its moves to B. To migrate these objects, A's TM must lock them during the move process. Deadlock is not possible because multiple objects are locked only while moving objects and concurrently committing transactions move disjoint sets of objects; for updating an object, locks are acquired one object at a time. Thus, a transaction S that is moving object x can only be waiting for a transaction T that is modifying x and T cannot be waiting for any other transaction. This implies that a cycle cannot occur in the waits-for graph.

Figure 5-6 shows the changes made to the commit protocol for supporting migration. Numbers indicate the order of messages, *i.e.,* message^i precedes message^{i+1}. Messages with the same numbers can be sent in parallel. Move-inform messages are sent by the destination server after it has installed the moved objects.

5.3 Blind Moves

In this section, we analyze the issue of blind moves. We first discuss the semantics associated with blind moves and then point out the implementation problems that arise in order to support them.

5.3.1 Semantics of Blind Moves

If blind moves are permitted, the user gives an object's new destination and the system does not execute an implicit locate-object operation for that object. This implies that an object that is being moved has not been necessarily located. Allowing blind moves has the desirable semantics that transactions are not aborted due to concurrent moves. Let us consider an example to understand the flexibility achieved by allowing blind moves.

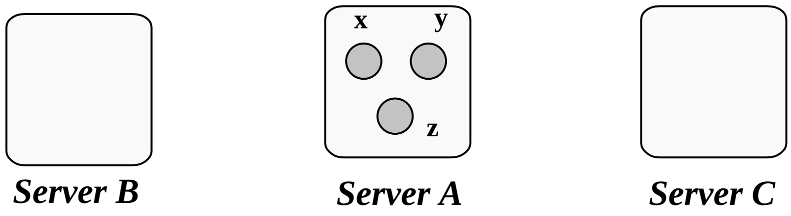
Suppose objects x, y and z are located at server A. Transaction S tries to move all these objects to server B and and transaction T tries to move y and z to server C (see Figure 5-7(a)). Both S and T pass validation and are committed; neither S or T is aborted due to concurrent moves. The order in which S and T are serialized determines the final destination of the objects. If S is serialized before T, objects y and z migrate to server C and object x at server B. If T is serialized before S, all the objects end up at server B. Figure 5-7(b) shows the situation in which S has been serialized before T.

Now let us assume that blind moves are not allowed. Since implicit locate-object operations have been executed for all the objects, either S or T must be aborted (moves and locates are conflicting operations). Figure 5-7(c) shows the scenario in which transaction S is aborted.

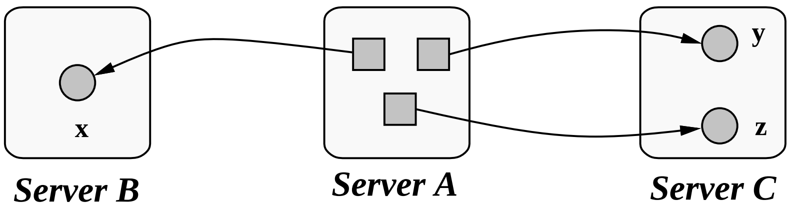
Thus, disallowing blind moves results in aborts due to concurrent moves. If blind moves are permitted, concurrent moves are ordered according to the timestamp order of their transactions. Blind moves are more intuitive to the user; there is no reason why a transaction must be aborted because an object has moved from the site as observed by the system at some point of time. If blind moves are permitted, the system can locate the object's location during the commit protocol and move the object to the desired destination.

5.3.2 Implementation of Blind Moves

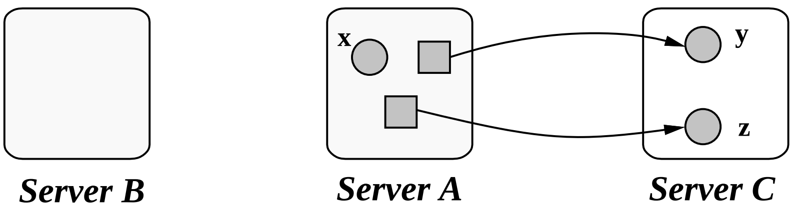
Allowing blind moves has the desirable semantics of permitting concurrent moves on an object but there are some implementation problems associated in supporting blind moves.



(a) Transaction S tries to move x, y, z to Server B
Transaction T tries to move y, z to Server C



(b) Blind moves allowed: S has been serialized before T.



(c) Blind moves disallowed: S has been aborted.

Figure 5-7: Object migration with and without blind moves

Blind moves complicate the validation and installation phases of the commit protocol. Consider the following scenario:

Suppose S and T are two prepared transactions on site A that are moving object x to servers B and C respectively. If a transaction U that has modified x tries to validate at server A, the TM must ensure that sufficient space is available at servers B *and* C. This is so because the system must be prepared to commit U irrespective of whether S and T are committed or not. Thus, at U's validation time, server A may have to send space-reserve messages to B and C. In general, if an object at server A is being moved by prepared transactions to n different servers, server A may have to send space-reserve messages to all the n servers. Not permitting blind moves ensures that at most one space-reserve message has to be sent for an object that is being modified by a validating transaction.

Permitting blind moves complicates the installation phase for object moves also. Suppose transaction T is trying to move object x from server A to B and is prepared at server A. While T is still in the prepared stage, another transaction U moves x from server A to server C. When T receives a commit message from its coordinator, it tries to move object x to server B. Since x has moved to server C, T's commit message must be forwarded to server C (which will try to move x to B). This situation is similar to the one discussed in Section 5.2.1 (scenario 2) where the commit message must be forwarded to the current site of an object that is being modified.

Another issue has to be handled in the installation phase. Suppose that two prepared transactions S and T are moving object x from server A to B and C respectively. Suppose S's timestamp is less than T's timestamp and T commits at A before S does; object x is moved to server C. When S's commit message reaches A and is forwarded to C, the latter server must ensure that x is *not* moved to server B. This is required because a later move of x (by T) has "absorbed" an earlier move of x (by S). Moving x to B now would be incorrect. Thus, if two transactions are moving the same object, the system has to ensure that the object's ultimate destination is the server specified by the transaction that has been serialized later. This condition is similar to Thomas's write rule [Bern87] for blind writes.

From the preceding discussion, we can see that supporting blind moves does complicate the commit protocol. To avoid these complications, our design disallows blind moves. As stated earlier, this will prevent concurrent move operations to be executed on the same object. However, since concurrent moves on an object are expected to be rare, disallowing blind moves will not be a serious restriction and therefore it is not worthwhile to implement the machinery needed to support them.

5.4 Summary

In this chapter we discussed a scheme for migrating objects among servers. We presented the primitives available to an application for locating and moving objects. In our design, migration has been made orthogonal to read/write operations by partitioning an object's logical state into two independent parts — the value-state and the location-state. Transaction semantics are provided with respect to the migration primitives. Atomic movement of objects is guaranteed so that either all or none of the objects are migrated; partial migration does not occur. Migration primitives executed during a transaction are also serializable with respect to each other. Providing transaction semantics with respect to migration makes it easier for users to reason about their programs. Furthermore, modeling the location-state in a way similar to the value-state allows us to implement object migration by adapting the validation schemes and optimizations presented in the earlier chapters.

We have presented the changes that must be made to the commit protocol for supporting migration; extra foreground and background messages may have to be sent in certain situations. We also discussed the semantics of blind moves and showed that although they offer reasonable semantics, our design does not support them because of implementation concerns.

Chapter 6

Conclusions

In this thesis, we have presented the design of a new transaction mechanism for a distributed client-server system. We have also described a facility to migrate objects among servers. In this chapter, we summarize our work and also present interesting problems for future research.

6.1 Summary

Our design for transaction management is based on optimistic concurrency control. We chose optimism over pessimism because we assumed that are few conflicts on objects. In such workloads, optimistic schemes tend to perform better than pessimistic schemes since they make better use of client caching and prefetching than pessimistic schemes. We have taken advantage of system characteristics to reduce the space and time overheads of the concurrency control mechanism.

We have also developed a mechanism to allow applications to migrate objects among servers. Our object migration scheme has been integrated with the concurrency control mechanism; this strategy allows us to provide transaction semantics with respect to the migration primitives also.

Our work has been done in the context of the object-oriented database Thor. We have made certain trade-offs in our schemes based on the environment we expect Thor to be operating in, *e.g.,* we assume that loosely synchronized clocks are available in the system. Based on such assumptions we have designed our schemes to optimize the normal case processing.

6.1.1 Concurrency Control

We first presented a validation scheme in which at most one transaction was allowed to validate or install objects at any given time. We later optimized this algorithm to allow multiple transaction validations and object installations to proceed simultaneously. This strategy reduces the validation delay since it prevents an incoming transaction from being blocked by other validating transactions. Allowing object installations to proceed in parallel with transaction validation is especially important for object migration; it is undesirable to delay a validating transaction while an object is being moved from one server to another.

In general, all optimizations suggested in this thesis aim at reducing the delay observed by an application. For example, the coordinator log protocol and short-circuited prepare reduce the number of messages in the first phase of the protocol; both schemes optimize the commit protocol and can be used in pessimistic systems also. Some of these optimizations take advantage of system characteristics such as loosely synchronized clocks, high availability of servers, etc. Note that these schemes reduce the commit delay time as observed by an application and also result in decreasing the total time taken to execute the commit protocol. As a result, we expect transaction throughput to increase due to these optimizations. However, some of these optimizations complicate and slow down the crash recovery process.

Our optimistic scheme uses backward validation for checking a transaction's operations. As discussed in Chapter 3, this choice was primarily made for efficiency reasons. However, backward validation does offer weaker semantics compared to forward validation because an application can observe inconsistent states of the database. Application programmers must be aware of this fact and program accordingly. The designers of Gemstone [Maier86] did not find this problem to be severe.

The asynchronous commit strategy suggested in this thesis is useful for applications that usually expect their commits to succeed. This facility is in consonance with the basic Thor philosophy of optimizing the common case; if transactions usually succeed in committing, it is wasteful for an application to wait for the result. In our design for asynchronous commit, an application does not know at most one transaction's result at any given time; to simplify the interface and the implementation, we did not allow more than one transaction to be pending at the frontend. An important consideration in using asynchronous commit is the change in code structure required to achieve the desired semantics. In Chapter 4, we presented an example to demonstrate the use of asynchronous commit. Depending on the needs of the application, the programmer may have to restructure his code.

6.1.2 Object Migration

Our work in object migration is different from previous research in the area of process and object migration because we have integrated our migration strategy with the concurrency control mechanism. Providing transaction semantics for migration makes it easier for users to reason about their programs. Some other design decisions that we have taken are motivated by the same reason. For example, migration has been kept orthogonal to reads/writes; this strategy also helps in avoiding some unnecessary aborts due to false conflicts. Not supporting blind moves is the only design decision made because of implementation rather than semantic concerns; allowing blind moves complicates the commit protocol considerably. In our design we have assumed that applications move objects rarely and information about migrating objects is propagated quickly to the relevant frontends and servers; our scheme takes these characteristics into account and ensures that normal case processing is not penalized because of migration.

Applications may attempt to use the migration facility for conflicting reasons. For example, one application may want to move a set of objects to one server whereas another may want to spread the same set of objects across the system. The migration mechanism will move objects according to the serialization orders of the application transactions; some higher level access control scheme is required to give priority to the relevant application.

An application programmer may not want work on the value-state to be aborted if a transaction passes the validation for the value-state but fails on the location-state (and vice-versa). That is, he may want to commit the changes to the location-state and the value-state independently. To support this requirement, the application interface needs to be modified; primitives have to be added to allow an application to commit operations of the value and location states independently.

6.2 Future Work

Our transaction management and object migration schemes can be extended to suit the relevant operating environment. Some of the assumptions we have made may not hold in those settings. This section discusses some of these issues and gives possible directions for future research.

6.2.1 Transaction Support for Application Multithreading

In our design, an application executes only one transaction at a time. If an application is multithreaded, it has to coordinate its threads as part of a single transaction. Users may prefer to execute multiple transactions in parallel. In the current design, they have to start multiple frontends for the same application. However, this approach is expensive since it leads to excessive duplication of data; it is also difficult for the programmer to coordinate these transactions. It is more desirable to have a scheme in which a single frontend allows an application to execute multiple transactions simultaneously. The frontend must be extended to support concurrent transactions. The application interface has to be modified so that the application can refer to different transactions; transaction objects can be used for this purpose. The application needs some way of associating threads with transactions. One possible strategy is to associate each thread with a different transaction. However, this mechanism may be overly restrictive since a user might want to coordinate a group of threads as part of a single transaction.

To allow an application to execute multiple transactions concurrently, a frontend must have some concurrency control mechanism for coordinating the access to its objects. A pessimistic or an optimistic scheme can be chosen for this purpose. For a pessimistic scheme such as locking, a lock manager must be implemented as part of the frontend. For an optimistic scheme, the frontend has to make a copy of an object before it allows a transaction to read or modify that object. The problem with this approach is that it decreases the effective size of the frontend cache. Note that at the end of a transaction T, the frontend can run a centralized validation algorithm to validate T and if T passes validation it can send T's information to the servers. Another strategy for the frontend could be to directly send T's information to the servers. The former approach adds complexity to the frontend. However, it has the advantage that it relieves the servers of some work. For example, if transaction R from frontend A aborts another transaction U at A, no message has to be sent for U to any of the servers since U will be aborted at A itself.

6.2.2 Hybrid Approach for Long-running and High-conflict Transactions

Our optimistic scheme is not well-suited for serializing long transactions; if a conflicting transaction invalidates a long-running transaction's operations, all the work done by the latter is wasted and has to be redone. It may happen that such a transaction is not able to run until completion because it keeps getting aborted by other committing transactions that invalidate its operations. Our scheme has to be extended to handle long-running transactions. One possible approach is to use pessimism for long-running transactions. At the beginning of a transaction, an application declares that it is going to execute a long-running transaction. During the transaction's execution, the frontend acquires locks on its behalf; the no-wait locking scheme can be used for this purpose. The validation algorithm has to be modified to handle locks; validating transactions that are not serializable with long-running transactions can be aborted.

As stated in Chapter 1, optimistic schemes achieve lower throughputs compared to pessimistic schemes in high-conflict environments. To achieve good performance in such environments locking can be used. However, for low-conflict environments, optimistic schemes perform better than pessimistic schemes. Thus, a mechanism is needed that adapts dynamically according to the characteristics of the workload. Gruber [Gruber94] is exploring such a strategy. He proposes a hybrid approach in which the decision to use pessimistic or optimistic concurrency control is done on a per-object basis and this selection is done dynamically. If optimism is being used for a particular object x and it results in excessive aborts, the system considers that object to be "hot" and starts using locks for x. When very few transactions conflict on x, the system switches back to using optimistic concurrency control for x.

6.2.3 A Utility for Reconfiguring Object Placement

In this thesis, we presented a mechanism that allows applications to migrate objects among servers. However, certain decisions about migration have been left to the application, *e.g.,* which objects have to be migrated, which server must they go to, when should they be migrated, etc. It would be interesting to design a utility that tries to make intelligent decisions based on different characteristics of the system and migrates objects. Previous work done in the area of load-balancing [Walds92] and process migration [Douglis91] can be used for designing this facility. This utility can monitor different aspects of the system and reconfigure the object placement accordingly. For example, it can determine which server is excessively loaded and with the help of the object usage pattern, migrate objects to lightly-loaded servers. It can also determine the network load between the application site and the servers and move objects accordingly. The fact that migration is orthogonal to reads and writes will ensure that unnecessary aborts do not occur during the load-balancing process.

6.2.4 Supporting High Mobility

Our migration scheme has been designed for an environment where movement of objects is rare. It would be interesting to support highly mobile objects. An important consideration in such an environment will be to avoid long surrogate chains from forming. If an object x keeps migrating from one server to another, remote references to x must be updated at a reasonably fast rate; otherwise, long surrogate chains may degrade the application's performance substantially. Also if different applications are trying to migrate a certain set of objects simultaneously, it can lead to excessive aborts. This high-contention problem can be solved using a hybrid approach similar to the one suggested in Section 6.2.2. If an object x is moved excessively resulting in aborts, the system considers x to be highly mobile and uses locks for x's location-state; the locate and move primitives lock the location-state of an object in shared and exclusive modes respectively.

Another way of reducing aborts for highly mobile objects is to permit blind moves. Blind moves complicate the migration mechanism but they decrease the number of conflicting locates on highly mobile objects. Furthermore, if multiple transactions are moving an object blindly to different servers, the source server can just move the object to the destination specified by the transaction that is serialized last.

6.2.5 Different Granularities for Concurrency Control and Migration

Our design for concurrency control and migration is based on the fine granularity of objects. Our schemes can be extended to operate at different granularities of control. For example, if there is some way of grouping objects and naming them, concurrency control can be performed based on object groups. These groups can either be exposed to the application or be internal to the system. When a server receives an incoming transaction T, it can validate the transaction's operations at the coarser granularity of object groups. If an object x passes this check, the server does not need to carry out a fine granularity test for object x. For example, suppose that version numbers are assigned to object groups also. The server executes the version check for the object group of each ROS object. If an object passes this test, the server does not have to execute the version check for that object. Grouping objects together and validating at different granularities may turn out to be advantageous in a low-contention environment; if the objects accessed by a transaction belong to a small number of object groups, only a few coarse granularity checks have to be carried out. This strategy has another advantage as the following discussion shows:

In the Thor architecture, version numbers are stored with the objects. During the version check of transaction T, the version numbers of T.ROS objects have to be read from the server state. If these objects are not present in the server's cache, there will be disk delays during the validation process. This problem can be alleviated by using object groups; if a transaction passes its tests at the object group level, the version numbers for individual objects are not required. Thus, at validation time, a server's cache need not contain all the T.ROS objects; it just needs validation information about the relevant object groups. Object groups can be used to validate the migration primitives also. Furthermore, if an application can name and create object groups, it can move objects using these groups.

6.2.6 Performance Evaluation

We are currently implementing our transaction management scheme in Thor. The performance of our schemes can be evaluated by measuring the transaction throughput and the commit delay observed by an application for different workloads. It would also be interesting to observe the performance difference caused by some of the design decisions we have made. For example, it is important to evaluate the number of aborts caused due to the watermark checks; if there are too many such aborts, the validation algorithm needs to be modified so that the watermarks are updated less aggressively and a transaction entry is retained in the VQ for some time even after the transaction has committed. Another strategy could be to maintain a watermark for each object group, *i.e.,* keep timestamp information at a granularity lower than the server level.

The performance of short-circuited prepare can be compared to the normal prepare mechanism to determine if the complexity added to the commit protocol is worthwhile or not. Another strategy that needs to be evaluated is the early send mechanism. It would be useful to determine the benefits achieved from this optimization; early sending the data may not be worthwhile if lots of redundant data is being sent by the frontend. The effectiveness of asynchronous commit can be measured by comparing its transaction throughput with the synchronous commit case for different workloads.

We can also determine how transaction throughput is affected due to the interference of object migration with normal operations such as object fetch and commit. For example, the difference in the commit delay and the system load in the presence and absence of migration would indicate the overheads of migrating objects. Measuring transaction throughput before and after moving objects would be useful to determine the effectiveness of a reconfiguration.

Bibliography

[Adya94] ADYA A., GRUBER R., LISKOV B., MAHESHWARI U. *Efficient Optimistic Concurrency Control for Distributed Database Systems.* In preparation.

[Agarwal87] AGARWAL D., BERNSTEIN J., GUPTA P., SENGUPTA S. *Distributed Optimistic Concurrency Control with Reduced Rollback.* Distributed Computing, 2(1):45-59, 1987.

[Bern87] BERNSTEIN P., HADZILACOS V., GOODMAN N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading MA, 1987.

[Black90] BLACK A. P., ARTSY Y. *Implementing Location Independent Invocation.* IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, January 1990.

[Ceri82] CERI S., OWICKI S. *On the Use of Optimistic Methods for Concurrency Control in Distributed Databases.* Proceedings of the 6th Berkeley Workshop, pages 117–130, 1982.

[Day94] DAY M., LISKOV B., MAHESHWARI U., MYERS A. *References to Remote Mobile Objects in Thor.* ACM Transactions on Programming Languages and Systems, 1994. To appear.

[Day94] DAY M. *Client Caching in an Object-Oriented Database.* Ph.D. thesis, Massachusetts Institute of Technology, 1994. Forthcoming.

[DeWitt84] DEWITT D. J. ET AL. *Implementation Techniques for Main Memory Database Systems.* Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 1-8, June 1984.

[Douglis91] DOUGLIS F., OUSTERHOUT J. *Transparent Process Migration: Alternatives and the Sprite Implementation.* Software — Practice and Experience, Vol. 2(18), 757-785, August 1991.

[Duchamp89] DUCHAMP D. *Analysis of Transaction Management Performance.* Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, pages 177-190, December 1989.

[Fishman84] Fishman D. H., Lai M., Wilkinson *Overview of the Jasmin Database Machine.* Proceedings of the ACM SIGMOD Conference on Management of Data, 1984.

[Franklin92] Franklin M., Carey M. *Client-Server Caching Revisited.* Technical Report 1089, Computer Sciences Department University of Wisconsin – Madison, May 1992.

[Gifford83] Gifford D. K. *Information Storage in a Decentralized Computer System.* Technical Report CSL-81-8, Xerox Corporation, March 1983.

[Gray79] Gray J. N. *Notes on Database Operating Systems.* In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, Chapter 3.F, pages 394–481, Springer-Verlag, 1979.

[Gray93] Gray J. N., Reuter A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc., San Mateo CA.

[Gruber89] Gruber R. E. *Optimistic Concurrency Control for Nested Distributed Transactions.* S.M. thesis, Massachusetts Institute of Technology, 1989.

[Gruber94] Gruber R. E. *Temperature-Based Concurrency Control.* Ph.D. thesis, Massachusetts Institute of Technology, 1994. Forthcoming.

[Häerder84] Häerder T. *Observations on Optimistic Concurrency Control Schemes.* Information Systems, 9:111–120, June 1984.

[Hagmann87] Hagmann R. *Reimplementing the Cedar File System Using Logging and Group Commit.* Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, pages 155-162, November 1987.

[Herlihy90] Herlihy M. *Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types.* ACM Transactions on Database Systems. Vol. 15, No. 1, pages 96-124, March 1990.

[Jul88] Jul E., Levy H., Hutchinson N., Black A. *Fine-Grained Mobility in the Emerald System.* ACM Transactions on Computer Systems. Vol. 6, No. 1, pages 109-133, February 1988.

[Kung81] Kung H. T., Robinson J. T. *On Optimistic Methods for Concurrency Control.* ACM Transactions on Database Systems 6:213–226, January 1983.

[Lai84] Lai M., Wilkinson W. *Distributed Transaction Management in Jasmin.* Proceedings of the Tenth International Conference on Very Large Data Bases, pages 466-470, August 1984.

[Lamb91] Lamb C., Landis G., Orenstein J., Weinreb D. *The ObjectStore Database System.* Communications of the ACM, Vol. 34, No. 10, October 1991.

[Lamport78] LAMPORT L. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM. Vol. 21, No. 7, pages 558-565, July 1978.

[Lampson93] LAMPSON B., LOMET D. *A New Presumed Commit Optimization for Two Phase Commit.* Technical Report CRL 93-1, Digital Equipment Corporation. Cambridge Research Laboratory, Cambridge, February 1993.

[Lausen82] LAUSEN G. *Concurrency Control in Database Systems: A Step Towards the Integration of Optimistic Methods and Locking.* Proceeding of the ACM Annual Conference, pages 64-68, October 1982.

[Levy91] LEVY E., KORTH H. F., SILBERSCHATZ A. *An Optimistic Commit Protocol for Distributed Transaction Management.* Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 88-97, May 1991.

[Lindsay84] LINDSAY, B. G., HAAS, L. M., WILMS, P.F., AND YOST, R. A. *Computation and Communication in R*: A Distributed Database Manager.* ACM Transactions on Computer Systems, 2(1), February 1984.

[Liskov84] LISKOV B. L. *Overview of the Argus Language and System.* Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, Cambridge, MA, February 1984.

[Liskov91] LISKOV B. L. *Practical uses of synchronized clocks in distributed systems.* Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, August 1991.

[Liskov91a] LISKOV B. L., GHEMAWAT S., GRUBER R., JOHNSON P., SHRIRA L., WILLIAMS M. *Replication in the Harp file system.* Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, October 1991.

[Liskov93] LISKOV B., DAY M., SHRIRA L. *Distributed Object Management in Thor.* In M. Tamer Özsu and Umesh Dayal and Patrick Valduriez, editors, Distributed Object Management. San Mateo, California, 1993.

[Lomet93] LOMET D. *Using Timestamping to Optimize Two Phase Commit.* Proceedings of the Second International Conference on Parallel and Distributed Systems, pages 48-55, January 1993.

[Mah93] MAHESHWARI U. *Distributed Garbage Collection in a Client-Server, Transactional Persistent Object System.* S.M. thesis, Massachusetts Institute of Technology, 1993.

[Maier86] MAIER D. ET AL. *Development of an Object-Oriented DBMS.* Proceedings of Oopsla-86, Sigplan Notices, Vol. 21, No. 11, Pages 472-482,November 1986.

[Mills88] MILLS D. L. *Network Time Protocol (Version 1): Specification and Implementation.* DARPA-Internet Report RFC 1059, July 1988.

[Mohan83] MOHAN C., LINDSAY B. *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions.* Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, pages 76–88, ACM, August 1983.

[Mohan86] MOHAN C., LINDSAY B., OBERMARCK R. *Transaction Management in R* Distributed Database Management System.* ACM Transactions on Computer Systems, Vol. 11, No. 4, pages 378-396, December 1986.

[Moss81] MOSS J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing.* Technical Report 260, MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

[Moss90] MOSS J. E. B. *Design of the Mneme Persistent Object Store.* ACM Transactions on Information Systems, 8(2), pages 103-139, April 1990.

[Mull85] MULLENDER S. J., TANENBAUM A. S. *A Distributed File Service Based on Optimistic Concurrency Control.* Proceedings of the Tenth ACM Symposium on Operating Systems Principles, pages 51-62, December 1985.

[Oki85] OKI B., LISKOV B., SCHEIFLER R. *Reliable Object Storage to Support Atomic Actions.* Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Decemeber 1985.

[Oki88] OKI B., LISKOV B. *Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems.* Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing, August 1988.

[Powell83] POWELL M. L., MILLER B. P. *Process Migration in DEMOS/MP.* Proceedings of the Ninth ACM Symposium on Operating Systems Principles, pages 110-119, October 1983.

[Rahm90] RAHM E., THOMASIAN A. *A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking.* Proceedings of the 10th International Conference on Distributed Computer Systems, 1990.

[Reed83] REED D. P. *Implementing Atomic Actions on Decentralized Data.* ACM Transactions on Computer Systems, Vol. 1, No. 1, pages 3-23, February 1983.

[Sinha85] SINHA K. M., NANDIKAR P.D., MEHNDIRATTA S. L. *Timestamp based certification schemes for transactions in distributed database systems.* Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 402-411, 1985.

[Skeen81] SKEEN D. *Nonblocking Commit Protocols.* Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 133–142, May 1981.

[Spector87] SPECTOR A. Z. ET AL. *Camelot: A Distributed Transaction Facility for Mach and the Internet – An Interim Report.* Technical Report CMU-CS-87-129, Deparment of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1987.

[Stamos89] STAMOS J. W. *A Low-Cost Atomic Commit Protocol.* Research Report RJ7185, IBM, San Jose, CA, December 1989.

[Tam90] TAM V., HSU M. *Token Transactions: Managing Fine-Grained Migration of Data.* Proceedings of the Ninth Symposium on Principles of Database Systems, pages 344-356, April 1990.

[Theimer85] THEIMER M. M., LANTZ K. A., CHERITON D. R. *Preemtable Remote Execution Facilities in for the V-system.* Proceedings of the Tenth ACM Symposium on Operating Systems Principles, pages 2-12, December 1985.

[Walds92] WALDSPURGER C. ET AL. *Spawn: A Distributed Computational Economy,* IEEE Transactions on Software Engineering, Vol. 18, No. 2, February 1992.

[Wang91] WANG Y., ROWE L. A. *Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture.* Proceedings of ACM SIGMOD International Conference on Management of Data, pages 367-376, June 1991.

[Weihl87] WEIHL W. *Distributed Version Management for Read-Only Actions.* IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January 1987.

[Weinreb88] WEINREB D. ET AL. *An Object-Oriented Database System to Support an Integrated Programming Environment.* IEEE Database Engineering Bulletin, 11(2):(33-43), June 1988.

[Zayas87] ZAYAS E. R. *Attacking the Process Migration Bottleneck.* Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, pages 13-24, November 1987.