BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M20

" Concurrency Control and Transaction Management in Observer2"

by

Adam R. Stauffer

Concurrency Control and Transaction Management in Observer2

Adam R. Stauffer

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

May 1993

This research project by Adam R. Stauffer is accepted in its present form
by the Department of Computer Science at Brown University
in partial fulfillment of the requirements for the degree of Master of Science.

Stanley B. Zdonik

Professor Stanley B. Zdonik
Advisor

6/15/93

Date

Concurrency Control and Transaction Management in ObServer2

Adam Stauffer

This document describes the design and implementation of an optimistic concurrency control system for ObServer2, a high performance distributed object store. The system supports user transactions by offering three services: object versioning, intention logging, and server coordination.

1.0 Introduction

Concurrency control is one crucial factor in the effectiveness of an object storage system which services more than one client. Multiple users can introduce nondeterminacy into each others work if there is no method of negotiating concurrent accesses to objects. Transactional concurrency control has been the traditional solution to this problem; the user brackets a set of updates and the storage service guarantees the success or failure of the all operations within the set. The client can retry operations if the transaction fails.

The Observer2 storage service uses the transaction model to allow multiple users to access objects in the system. The concurrency control facilities implement an optimistic scheme in which the correctness of the updates are not verified until the end of the transaction. This method reduces message traffic and therefore reduces execution time. The implementation is fairly simple in the general case: all objects in the system have associated version numbers. Locking is "lazy;" any client can use any object it has cached and conflicts are determined when the transaction commits based on the objects' version numbers. The concurrency control manager maintains the set of correct object version and tracks transactions as they occur. When a client signals the end of a transaction these stored version numbers are compared with the cached copy version numbers.

A concurrency control service has additional concerns of hardware or network failure and latency. Server coordination can become difficult given the nondeterministic nature of network communication and the real time constraints of a database system. The concurrency control system must maintain a balance between the desire to wait for a message and the desire to make forward progress. Host failure must also be considered so a transaction can properly terminate.

1.1 ObServer2

A general understanding of the ObServer2 architecture is helpful to understand fully the design of the concurrency control system. This section gives a brief overview of the architecture; for a full description see [lang93].

Observer2 is a distributed object storage system which allows multiple clients to access objects stored at many distributed servers. The system supports methods on a limited set of fairly mature abstract data types such as B+trees and segments but is fully configurable to allow additional types to be built into the system as deemed necessary.

The store is comprised of several components: a transaction management subsystem, a logging subsystem, a memory management module, and several class modules. These modules interact using a set of defined interfaces so the construction of additional storage classes is possible. The core system maintains naming, transactional concurrency control, and storage of the objects while the class provides the methods required for that class. The main interface to clients is through the class interface for object methods and through the Transaction Manager for concurrency control.

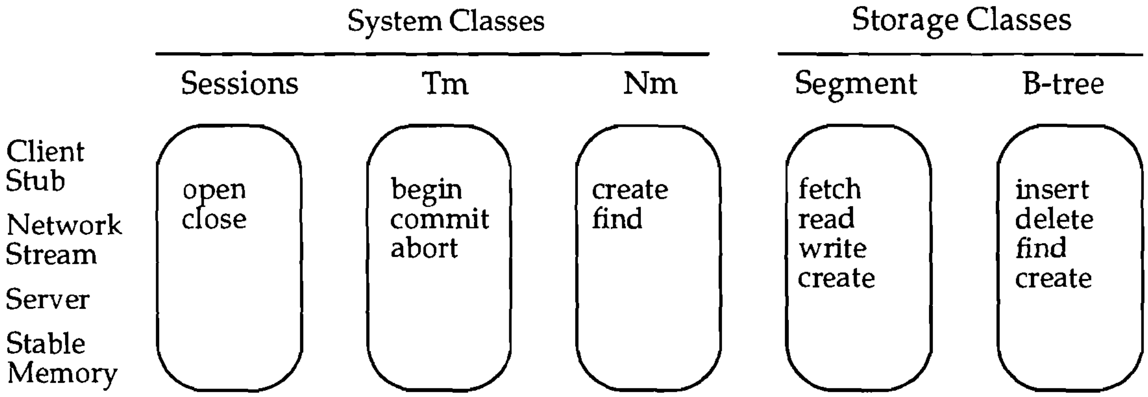


Figure 1: The Observer2 architecture.

2.0 Requirements

The concurrency control system exists primarily to support user transactions and insure the correctness of updates to the object store. Within this general goal the requirements can be broken into four parts:

- management of executing transactions
- maintenance of object versions
- coordination of multiple servers
- robust support for delays and failures

2.1 Tracking Executing Transactions

An Observer2 server can be providing objects for many clients, each involved in separate transactions. The server must track the state of each transaction it is a participant in as well as the operations each client is performing. These operations are reconciled at the time of a commit. The concurrency control system should be able to quickly determine if a transaction is properly ordered and whether or not it can succeed given knowledge of the previously committed transactions and the operations performed during the transaction.

2.2 Object Versions

When a transaction is about to commit there must be a method of determining whether or not any accesses were to inconsistent data. An object cached at a client might be updated at the server by another host; if the cached copy is used in a subsequent transaction then it may be inconsistent with the correct state of the system. Each object should therefore have a version number and the concurrency control system should know the must recent committed version of each object cached.

2.3 Server Coordination

One transaction may involve objects from many different servers. The concurrency control system on each server must be able to cooperate to determine whether the global operations can succeed based on each local state. This operation must be recoverable to insure the correctness of the database given the possibility of network or host failure.

2.4 Recovery

The system must be fault-tolerant and able to recover from system crashes, network failures, and media loss. Any transactions that were committed must be resilient to failure. Executing transactions should be able to continue where they left off after the last logged operation.

3.0 Design

The Transaction Manager is comprised of several major components each described in detail below. Sharing the client's address space is a set of stubs which communicate with the Observer2 server. The server module is comprised of a transaction and object version

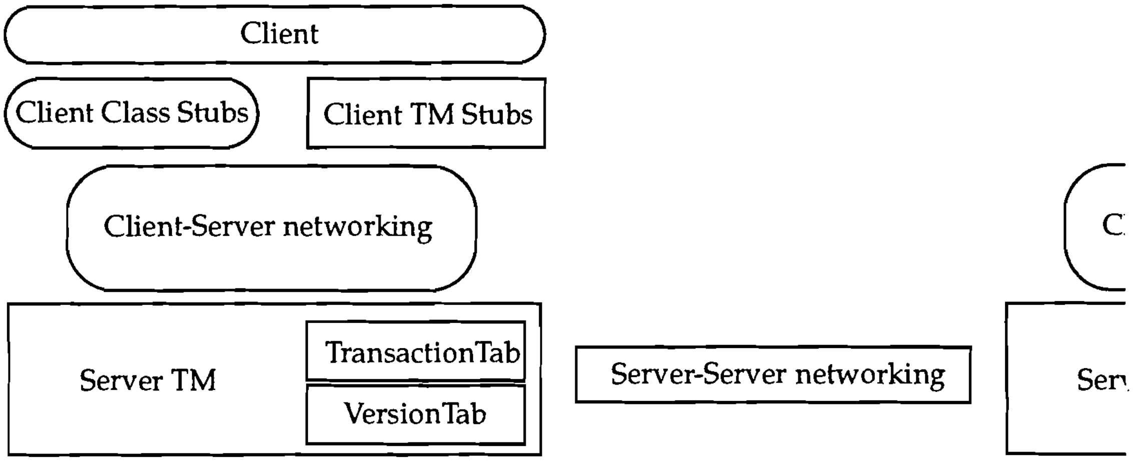


Figure 2: The basic outline of the Transaction Manager. Tm components are shown as rectangles. Transaction Identifiers

table as well as a facility for communicating with other TM's (figure 2).

3.1 Client Stubs

The client stubs encapsulate all TM functionality from the user into a simple transactional model consisting of begin, commit, and abort operations. When a client calls begin to start a transaction the TM notifies each registered class with a transaction identifier to associate to data accesses (3.1.1). The classes then notify the client TM of any objects referenced and these messages are logged to stable storage at the server (3.1.2). Finally when a transaction commit request occurs the client TM executes the first step of the commit protocol (3.1.3). The stubs return the result of the transaction to the client.

3.1.1 Transaction Identifiers

Each executing transaction has associated with it an identifier, a Transaction ID (TID). Each TID is composed of two long words, a local time field and a host number, which together guarantee the TID's are unique throughout Observer2 (figure 3). The TID constructor guarantees that no two TID's on the same host have duplicate local time fields while host numbers are the tie-breaker between TID's generated at the same time on different hosts. A TID is generated when the client TM receives a begin transaction request. Each class stub on the host is then notified of the TID and that a transaction has begun. The TID follows the transaction until completion; it is forgotten by the client TM once the final outcome of the transaction has been learned.

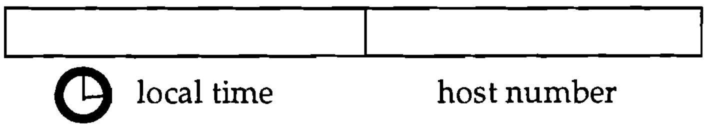


Figure 3: Transaction Identifiers and timestamps

Each transaction also has associated with it a timestamp (Ts). The timestamp is generated by the client TM when a commit request arrives. This timestamp allows the server TM to order transaction requests consistently at different locations. To guarantee unique timestamps system wide they are constructed in the same manner as TID's.

3.1.2 Intention Logging

The server TM must insure two transactions are not both allowed to commit if they have conflicting operations. The server TM therefore needs knowledge of which objects were referenced during which transaction, and the manner of these operations. All updates to the store flow through the TM and are noted with the transaction identifier. The storage of this information is accomplished by use of the Observer2 logging system [reil93]. The client TM exports a method to the classes which allow them to log an arbitrary length byte stream encoding the operations on an object. This is coupled with an object identifier and the version number of the referenced object. Several of these operations can be bundled together and logged in one step to reduce the amount of messaging in the system. At the conclusion of a transaction these operation records are sent back to the class at the server to determine if the operation can be correctly applied within the server's context.

While intentions are being logged the client TM is also collecting a list of servers from which objects have been fetched. Each log operation makes certain the server to which the request is being sent is on this list. The list of servers is used during the commit phase.

3.1.3 Commit

When a commit is requested the client TM sends the accumulated list of servers to each server on the list. This is the start of the two-phase commit protocol. The first server on the list becomes the coordinator of the transaction while the rest (if there are any) become participants. This message is the first step of the 2PC requesting the servers prepare for the transaction. The client then waits for a message on any of the connections for the result of the transaction and returns the result to the caller.

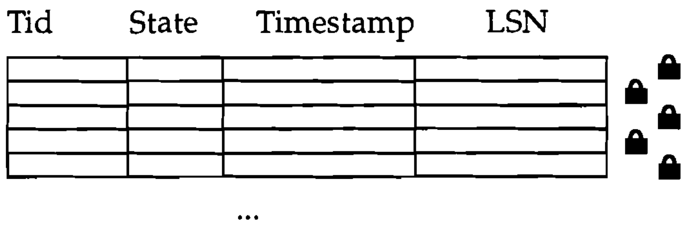
3.2 The Server TM

The server TM is the heart of the concurrency control system for Observer2. It tracks executing transactions and the operations that are occurring within them (3.2.1). It maintains the version table of objects to test the ordering and success of a transaction (3.2.2).

Finally it communicates with other servers to determine if the transaction can succeed in the global context (3.2.3).

3.2.1 The Transaction Table

The Transaction Manager maintains a list of its currently executing transactions in the transaction table (figure 4). This table has one entry for each outstanding transaction with a status of in progress, prepared or committed. Aborted entries are removed once all participants in the transaction have been notified. The table stores a timestamp which corresponds to the time of the commit at the client. Finally there is a log sequence number which has a different meaning depending on the table state (figure 4).



State	LSN value
In progress	Intentions ——► Intentions ——► ···
Prepared	Prepare Record ——► Intentions
Committed	Commit Record ——► Prepare Record —► Intentions

Figure 4: The transaction table and the row locks. The structure contains all of the information associated with a transaction(top). The LSN in the table points to a different log record depending on the state (bottom).

The LSN refers to the log record for the transaction and this record varies with the row state. Once a transaction begins and until the client signals its end the LSN in the table points to an intention record in the log. Intention records contain an LSN field so they can be strung together as a linked list. Once a client has ended a transaction there are two phases entered by the row (the commit algorithm is described in more detail below, but they are prepare and commit). First the transaction determines if it can commit based upon local information. The LSN refers to the log record recording the timestamp of the change to this state and the intention records which comprised the transaction. The second state occurs when the global system state has been negotiated and the transaction succeeds. Another record is logged with the final timestamp and a reference to the LSN of the prepare record.

The transaction table has some other supported features to insure transaction correctness. Each row in the transaction table can be locked and allows the lock to be held between lookup and update to insure the correctness of the returned data. The table also has locks on important data structures to allow multithreaded access to the methods. The table maintains a timestamp for the last transaction flushed from the table. It cannot accept any late transactions with timestamps before this one because it cannot determine the possibly conflicting set of objects referenced.

3.2.2 The Version Table

Each object in the system has a version number which is the LSN of the most recently committed copy of an object, or the operation that when applied to the copy of the object in the database partition which would create the most up-to-date version. Logging operations allows the server to increase throughput by applying operations asynchronously. See [lang93] for more information. The version number is used to verify the correctness of updates. This table is just like the transaction table in that it supports row locking, concurrent access and updates. Version numbers are updated when a transaction commits and the operation has been applied to the object.

3.2.3 Messaging

Due to the high overhead of initiating a TCP connection and the transient nature of a set of servers involved in a transaction the server TM use a lightweight UDP based messaging system. The system allows several threads to wait for a message on one port and be awoken when a message arrives addressed to that particular thread.

3.3 Multiple Servers

Since ObServer2 transactions can potentially involve objects from multiple stores, the Transaction Manager must be able to coordinate operations between these stores to insure correctness. The two-phase commit protocol[grey90] allows multiple servers to negotiate the outcome of a transaction. The general protocol is outlined in (3.3.1) while special cases are covered in the following sections.

3.3.1 The Protocol

Server synchronization is accomplished through the use of a two-phase commit algorithm. Each server determines independently whether their portion of the transaction can successfully complete and then votes on the transaction in the prepare phase (3.3.2). A coordinator tallies the votes and determines if the transaction is ordered properly or not (3.3.3). If all servers return positive responses then a consensus has been reached and the transaction will commit. Transactional concurrency control relies on the ability to order the application of operations at each server. If one server orders transaction T1 before T2 while another reverses this order then clearly one or both might be in error. This problem is solved with a timestamp mechanism. If a transaction arrives out of order late it still may be possible to retry with a later timestamp (3.3.4).

3.3.2 Prepare

Each server is notified by the client that a transaction will commit and that they should determine their local state. A thread is started to handle all messaging and operations for this transaction for its duration. The list of servers sent in this prepare message has the coordinator of the transaction listed first. The servers log to stable storage whether or not the transaction can continue and reply to the coordinator. Once the a positive vote for the transaction has been sent the participant must be prepared to commit the transaction so no conflicting operations can be applied before a result is received. With the vote the participants also send a list of transactions which were committed with later timestamps than that of the currently committing transaction. This conflict set is used to determine ordering.

At the coordinator a thread is started within the server to execute the messaging primitives and tally the results, also for the duration of the transaction. This coordinator simply tallies the votes of the participants and returns a message to commit if there is consensus or abort otherwise. It also uses the conflict set to determine if the transaction is properly ordered (3.3.3) and can abort a transaction if there is a conflict.

3.3.3 Transaction Ordering

It may be the case that a transaction prepare request arrives at a server after some number of transactions with later timestamps have been committed. These transactions are termed *late* because the servers expect operations to arrive in the order of their timestamps. There are rules the server can check to determine if the transaction can proceed.

Let *l* be a late transaction and T be the nonempty set of prepared or committed transactions with timestamps later than *l*. There are several situations the Transaction Manager might encounter.

1. *l* can proceed if the set of objects referenced by *l* has no objects in common with the sets of objects referenced by each transactions in T, or any common operations are reads.
2. *l* and $t \in T$ do not conflict if the transactions have no common remote participants.
3. *l* and $t \in T$ do not conflict if for all common remote participants *l* arrived after *t*.

The Transaction Manager can evaluate these clauses for a given transaction to determine whether or not the transaction can continue or must be aborted or retried.

3.3.4 Retry

When a transaction prepare message is determined to be late there are two possibilities for its resolution. The coordinator has a list of conflicting TID's which it can use to exercise rules 2 and 3 in the previous section. The participant checks rule 1 and could possibly return a new timestamp with its vote for the transaction. This new timestamp is a retry time and the transaction is retried with that value.

3.4 Recovery and Persistence

Persistence and recovery from failure is achieved through a checkpointing mechanism. Periodically the Logging subsystem requests that the Transaction Manager save its internal state. The version and transaction tables are then collected and flushed to the stable logging system. An atomic update to system metadata records the position of the new checkpoint. The recovery of the system after a failure then involves finding the last checkpointed version of the system and replaying the log activities to construct the latest consistent state. The Log Manager performs the replay and utilizes the recovery interface to the Transaction Manager to inform it of the new state.

4.0 Implementation

This section describes each modules interfaces in more detail.

4.1 The Architecture.

The transaction manager is composed of several classes (figure 5). Each class interface is described in detail in the following sections.

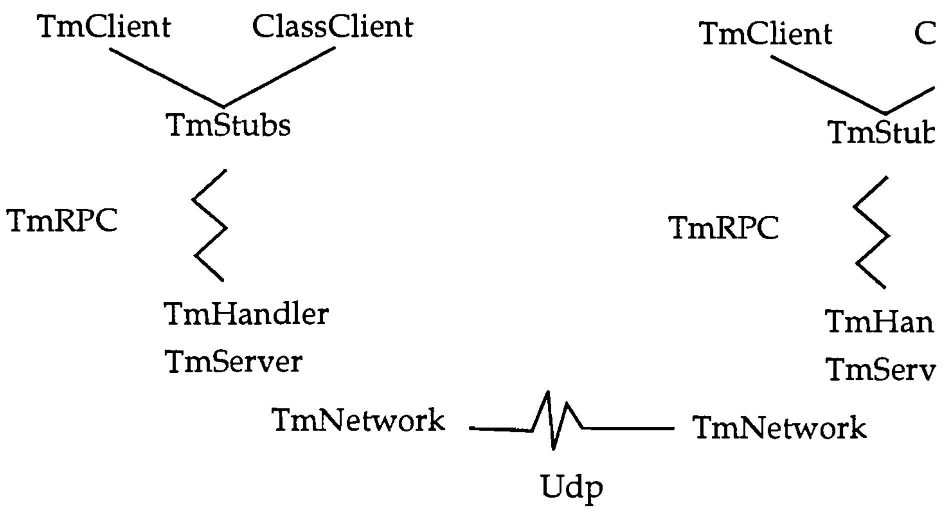


Figure 5: The general class description

There are two interfaces exported at the client side of the system, one to the client application (TmClient) and one to the classes built into the store (TmStubs). The client has the ability to begin and commit a transaction. The class can notify the TM of updates to objects by sending collections of operation records which are bundled and put into the log. The TmStubs module communicates via TmRPC to the server.

The TmServer module is given the task of tracking and logging the operations that arrive from the clients and determining the correctness of transactions as they occur. The TTab and VTab allow the management of transactions and object versions, respectively. the server receives messages from the client via a buffered TCP connection (Network-Streams). The connection medium is encapsulated in the TmRPC module which defines messages with send and receive methods. These messages are created, sent and received in the TmStubs and TmHandler.

The UDP messaging is accomplished with the TmNetwork subsystem sending messages defined in TmMesg.

4.1.1 Tid and Ts

These modules build transaction identifiers and timestamps, respectively. They are currently implemented identically; both have comparison methods and methods to send and receive over the NetworkStream. Both also provide preprocessor macros to marshall the representation into a structure which can be sent to the log. Given a structure TIDrawtid and a Tid the macros extract the representation into/from the raw representation by copy.

```
struct TIDrawtid {
    time_t local_time;
    unsigned int hostid;
};


#define RAWTOTID(RAW, TID)
#define TIDTORAW(TID, RAW)
```

4.1.2 TmClient

This module is linked into the client and exists simply to pass calls into the TmStub. A new transaction identifier is constructed when a begin transaction is called and a timestamp is constructed during a commit, but otherwise this module does not contribute to the operation of the transaction.

4.1.3 TmStub

The TmStub translates client calls into RPC calls to the server via NetworkStreams. It constructs messages defined in TmRPC and calls the send method on the objects.

4.1.4 TmRPC

TmRPC defines the messages that can be sent over the NetworkStream and encapsulates the sending and receiving from the rest of the Tm. If primary client-server messaging system is modified this package will need to be updated as well.

4.1.5 TmHandler

The TmHandler registers with the dispatcher the individual handlers for the Tm messages. These handlers are called when a message arrives to decode and dispatch the message to the appropriate methods within the TmServer. The result of the function calls are returned to the sender also via a message defined in TmRPC. The dispatcher creates a new thread within the server to handle the message.

4.1.6 TmServer

The TmServer class is the heart of the transaction system. It is the control for the rest of the code within the system.

```
Commit(Tid, Ts, SidList)
```

This method is the main public hook into the TmServer module. The real work of the transaction is performed in the following three methods. Commit updates the Transaction table with the result of the prepare and either participate or coordinate.

```
Prepare(Tid&, Ts&, TidList&, int loner, Ts& lastTs);
```

This method performs the prepare step of the 2PC. It looks into the transaction table for the provided Tid and walks the logged intentions. For each operation request within the set of intention records the appropriate class is called to determine the success of the operation. If the classes all can correctly apply the operations then the conflict set is built (by calling the TTab) and the ordering of the transaction is considered.

```
Participate (Tid&, Ts&, unsigned long c_addr, ClassServer::Sta-
tus, TmMesgBuf&);
```

If the server is a participant in the currently executing transaction then this method is called to perform the necessary communications. This is a simple routine because it just marshalls the messages and waits for a response from the coordinator.

```
Coordinate (Tid&, Ts&, SidList&, ClassServer::Status, TmMesg-
Buf&);
```

This method accepts the votes of participant servers and determines if there is a consensus. It also examines the intersection of the conflict sets for the same purpose. The participants are notified of the result of these operations and the appropriate log messages are applied.

Coordinate uses a LEDA dictionary to store the tallies from each participant. Prepare uses a LEDA stack to traverse the linked list of intentions in reverse.

4.1.7 TTab

These tables encapsulate the storage of executing transactions. A list of committed transactions sorted on timestamp is stored to allow quick construction of a conflict set when ordering transactions. Also a timestamp of the last deleted transaction is stored for the ordering of transactions. (See 3.3.3)

```
 int getRow ( Tid& tid, Lsn& lsn, TT_rowstate& st, Ts& ts);
 int setRow ( Tid& tid, Lsn& lsn, TT_rowstate newstate, Ts& ts);
 int unlockRow ( Tid & tid);
```

Rows are accessed and updated through the above interface. A lock is taken on a call to getRow and is not released until setRow or unlockRow is called. This is to allow correct access to the table data. If the row entered is a committed transaction then the committed list is updated.

```
 void conflict(Ts& ts, TidList& early, Ts& maxts);
```

Conflict returns the list of transactions (a LEDA list) which were committed before TS but had later timestamps than TS. It also returns the maximum timestamp of the set so an advisory can be sent to the late client.

6.0 References

[gray90] Gray, J.N. Notes on Data Base Operating Systems. In *Lecture Notes in Computer Science*, R. Bayer, R. Graham and G. Seegmuller, Eds., Springer-Verlag, 1978, pp 293-481.

[lang93] Langworthy, D.E. Observer2: Extensible High Performance Support for Persistence, *PhD. Thesis Proposal (Draft).*

[reil93] Reilly, P.A. Logging and Recovery in ObServer2.