Yale University
Department of Computer Science

How to Create a Failure Tolerant Distributed System[1]

Jonathan Hochman

YALEU/DCS/TR-799
June 1990

[1]Submitted in partial fulfillment of the BS/MS degree requirements, Yale University Department of Computer Science.

How to Create a Failure Tolerant Distributed System*

Jonathan Hochman

Abstract

Many distributed systems include a component which can be described as a transaction system. The problems associated with transaction systems are a major concern in the field of database design, namely how to recover transactions which may be interrupted by processor or communication failures, and how to serialize transactions in order to maintain desired transaction semantics. This paper introduces a transaction system protocol, the Four Phase Protocol, with a recovery mechanism that works in the presence of unlimited crash failures and message deletion failures. The Four Phase Protocol concentrates on simulating a failureless distributed system. Other problems are left to be solved by higher levels. Serialization, to name one problem, can be solved by carefully designing transaction requests, as opposed to building a solution into the execution mechanism.

1 Introduction and Motivation

A transaction is a discussion between two processes which is moderated by some set of protocols. "Transaction Systems" are an important area of database design because they are essential for making data more available, and for applications which are naturally decentralized. Two of the most significant problems in database design are serializing transactions, and recovering from crashes.

Some transactions may take a long time to complete, and it may be desirable to have many transactions occurring at once in order to improve transaction throughput. Unfortunately, some transactions may require a semantics which forbids other transactions from doing certain things concurrently. For example, if one transaction is transferring funds in a database of bank accounts, it would be undesirable to have another transaction concurrently summing up the totals

*Submitted in partial fulfillment of the BS/MS degree requirements, Yale University Department of Computer Science.

in all accounts. The second transaction might come up with the wrong answer. This problem is especially troublesome in a distributed database where various portions of the data may be under the control of different manager processes. Serialization is a way to make transactions look as if they execute in some serial order, one after the next. Interleaving may occur, but the serialization method only allows interleavings which do not disturb the property that all transactions must appear to happen in some consistent serial order.

In developing a new transaction system, serialization of transactions should not be required. In [3] it was shown that serialization may reduce the availability of information. In order to serialize transactions, it becomes necessary to restrict the order in which certain events occur, and this frequently decreases the speed at which transactions can be processed. Often, one transaction must wait for another to pass some critical point before proceeding. While serialization may be desired sometimes, it is not necessarily required at all times. Hence, it should not be part of the transaction system.

The transaction system should allow for the recovery of state, and continued processing of transactions in the event that processes crash or messages are lost. Recovery means reconstructing the local state of a process after it has crashed and restarted, and seeing that transactions in progress are completed as if the crash had not occurred. Ideally, any process in the transaction system should be able to fail at any time, restart, recover, and continue execution without altering the semantics of any possible transaction.

It is best to address orthogonal problems as separate issues. First, as a basis, a non-serializing transaction system is presented with a simple recovery algorithm that allows distributed transaction processing to proceed in the presence of failures. Next, a method for serializing transactions is demonstrated which can be built implemented on such a transaction system. The result is a clear and simple model for discussing the design of distributed transaction systems, including databases.

2 Transaction Systems

The name "Transaction System" should suggest a more general model than a name like "Database System". A transaction system is a set of protocols for running transactions on a group of distributed processes. This includes distributed databases, centralized databases (one manager process), any client-server application, and even a hardware device which might be shared by several processors. The problems associated with transaction systems can and should be dealt with separately from database problems such as serializing transactions. A transaction system does not have to provide serialization, and a database manager need not concern itself with recovery if it is built on a failure tolerant transaction system.

2.1 Distributed System Model

Assume that a transaction system is pictured as a network of nodes. Each node has a transaction interpreter, a dataspace for storing data objects, a log for recording transactions, and possibly a checkpoint structure which is used to save local state periodically in order to be able to recover state after a crash. Every node can freely communicate with any other node via channels. A channel is a reliable two way communication path, such a pair of TCP/IP sockets. A channel can crash at any time, in which case it may have to be reestablished. In the process, messages in the channel may be lost. Essentially, deletions can occur, but reordering and duplication of message is forbidden. Also, there is assumed to be a global clock accessible to every node.

A transaction system with multiple processes may have multiple dataspaces, logs and checkpoints. A distributed database which consisted of a large collection of data servers and lots of client processes would be an example.

The objects in dataspace may be relations and tuples in the case of a relational database, or they may be tables, I-nodes or any other structure. Objects might also be active. One could imagine a database of living objects, such as a ticking clock, or a coin which randomly flips itself every time someone asks it for its current value. Such a thing is not required, but would be accepted by this model.

The log is a special object which can be read or appended to, and which survives crash failures intact. A file on a disk drive which has read and write semantics which guarantee that it is not corrupted during a crash meets these requirements.

A checkpoint structure is like a log in that it must survive crashes, but the checkpoint is only read and written periodically. The differences and uses of checkpoints and logs are described in [2].

The transaction interpreter is a control algorithm which links the execution of transactions to changes in dataspace and the log. Periodically the interpreter may decide to create a checkpoint. If a node crashes, or a channel goes down, the interpreter is responsible for restarting whatever transactions were in process. The main result of this paper, the Four Phase Protocol, is used by the interpreter, running on a distributed system which suffers from failures, to simulate a failureless transaction system. The interpreter contains state because it knows the exact stage of its transactions in progress.

So in summary, the state elements of a node are its:

1. dataspace

2. log

3. checkpoint

4. transaction interpreter (transactions in progress).

2.2 Types of Transactions

Transactions may be of two different types. First, there are atomic transactions which are guaranteed to affect the state of only one node, which are executed on one node in an atomic fashion. Secondly, there are non-atomic transactions which may consist of a series of atomic transactions and intermediate computations. These commands generate transactions that affect more than one node, and these transactions may be interleaved in time with other transactions.

Let us distinguish between atomic transactions by calling them "actions" while referring to non-atomic transactions as "scripts".

Definition 1 *A* **transaction** *is the execution of a script or action. A transaction may be initiated by a transaction request which must specify either an action, or a script, and possibly some parameters which are fixed values (call by value).*

Transactions may be nested. A transaction may initiate and receive values from subordinate transactions. Such transactions are discussed in depth in [7].

Definition 2 *An* **action** *is an atomic transaction which only depends on the state of one node, and is executed on that node. All actions on a single node are executed serially on that node. Other actions may occur asynchronously on other nodes.*

The transaction system is required to serialize actions on each individual node. This can be done using the traditional methods of serialization. We may assume that actions are never executed concurrently, that each node has only one thread of execution which processes actions, or at least it makes things look that way.

Definition 3 *A* **script** *is a non-atomic transaction which may affect the state of many nodes, which is not guaranteed to be executed serially by the transaction system.*

2.3 Primitives, Including "Run"

Primitives are the programs or procedures which may individually, or in aggregate form a transaction. Primitives are the link between transactions, and alteration of dataspace.

For the purpose of illustrating a transaction system here, the following atomic transaction (action) primitives are assumed: *add*, *delete*, *find* and *match*. *Add* and *delete* merely insert or remove a specified object from the dataspace of a particular node. *Find* and *match* both accept a pattern, and return either one or all objects in a node's dataspace which match the pattern.

Actions are primitives. There is an additional primitive named *run*. *Run* takes arguments including the specification for a transaction, the name of a node, and perhaps some other parameters affecting execution such as timeout values *run* opens a subordinate transaction as specified, with the indicated node. This is a mechanism which scripts may use to perform actions on other nodes, or to call other scripts. *Run* itself is a script; it is the primitive script.

It is possible to create other primitives. They must be classified as either actions or scripts, and must adhere to the restrictions presented in this, and the next section, otherwise the results of this paper may no longer follow.

2.4 Limitations on Scripts

A script may be thought of as any program which includes actions. A script may also *run* subordinate transactions, like a program calling a subprogram. A script is expressly forbidden from gaining any information from any operation other than by performing an action, or running another script. Information may be supplied by arguments to the script, but these values must be fixed at the initiation of the transaction, just like a call-by-value procedure.

Thus a script cannot flip coins or read the system clock. It would be possible, however, to include an object in dataspace corresponding to the system clock, or to a flippable coin. The reason for this is to establish accountability. It is essential to be able to continue a transaction interrupted by a crash, and in doing so the Four Phase Protocol may rely on tracing the past history of a transaction, and the only recorded events are actions and scripts. These events are stored safely on the log, as we will see.

2.5 Actions vs. Scripts

Why do we need both actions and scripts? If an outside user, Sally, runs a script, she generates a series of actions along with some intermediate computations. The same outcome could be achieved if Sally executed the actions one by one, while performing any necessary intermediate computations on a pad of paper, or such. On the other hand, could she design a system which had lots of scripts, one for every desired type of transaction, and then skip the idea of having actions?

Actions and scripts are logically different because actions are never executed concurrently on the same node, while scripts may be interleaved, and may be executed concurrently. A script is different from a series of actions because once it is initiated, it is guaranteed to execute all the way to completion. If a user is initiating a stream of actions, there is no guarantee that he will not stop right in the middle. A script may leave the system in an transitional state while it is executing, but since the system has a correct recovery algorithm, the script will eventually complete, even in the presence of unlimited failures, and the desired final state will be reached.

Atomic transactions are necessary in any reasonable system. It would be highly undesirable if $delete(X)$ and $find(X)$ could be interleaved. The result of $find(X)$ might return total garbage. If interleavings are desired, the object X can be broken into two pieces, X_1 and X_2 which may be handled concurrently. On some level though, there is a conceptual atom of data which must be dealt with in one uninterrupted act, otherwise it would be impossible to have any sort of consistent transactions. If $delete(X)$ and $find(X)$ occur simultaneously on the same single object X, then $find(X)$ could return garbage.

3 Definition of Failure Tolerance

The transaction system is processing a stream of transaction requests. We would like the effect of this stream of requests in the presence of an arbitrary number of failures, namely nodes crashing and requests being lost, to be consistent with the effect of the same stream of requests in the absence of failures.

A stream of requests is generated by users outside the control of the transaction system, and by processes within the transaction system. The stream is defined to be a sequence of transaction requests ordered by the time that they are received by any process in the transaction system. In the event that multiple requests are received by multiple processes at the same tick in time, they are ordered in some arbitrary and deterministic fashion.

A stream of requests causes the transaction system to alter its global state. Two executions of the same stream of requests are consistent if the two outcomes, the two resultant global states, are the same, or if differences in the resulting states can be explained by differences in the way scripts are interleaved.

Let us say we have a stream of transactions T which consists of the following transactions, ordered by the global clock time at which they are opened.

$$A_1 S_1 A_2 A_3 S_2 S_3 A_4$$

All A_i are actions, and all S_i are scripts. If we take a transaction system in initial state q_0 and run T from the initial state x times, we may get x different final states $f_1, ..., f_x$. We desire a transaction system where all f_i are consistent.

Definition 4 *The* **global state** *in a transaction system is a tuple consisting of the states of the dataspaces of all nodes.*

Definition 5 *A* **transaction stream** *is a sequence of transaction requests, requests to initiate a transaction, ordered by the time which the requests are opened by nodes in the transaction system. A request is considered to be opened when it is written on the log of a node in the transaction system.*

Definition 6 *An* **execution path** *of a transaction stream is the partial ordering by global time of actions executed on all the nodes.*

The order in which actions are executed on a single node forms a total order, because they occur serially. Actions executed on different nodes are not synchronized, so they may be unordered by the execution. Hence, the execution path of a transaction stream is a partial ordering of actions based on when the actions are executed.

Every transaction is the execution of either an action or a script. A script can only affect the global state of the system by the actions which it contains. Hence the execution path of a transaction stream starting from a given initial state completely determines the final state.

A stream of transactions can generate many different execution paths because scripts may be interleaved arbitrarily, although all are assumed to complete by the end of the execution.

Definition 7 *Two global states, f_1 and f_2, are* **consistent** *if and only if there exist two execution paths, e_1 and e_2, of a single transaction stream, T, starting from a single initial state q, which respectively yield the final states f_1 and f_2. We say that f_1 and f_2 are consistent final states of stream T run from state q.*

Definition 8 *Let the function* $\mathbf{RESULT}(T, D, q) = F$ *where F is the set of final states reachable by all valid execution paths of stream T, on a transaction system D, starting at state q.*

Definition 9 *A transaction system, D, is* **failure tolerant** *if and only if for every possible transaction stream T and every possible initial state q, the states in the set $F = RESULT(T, D, q)$ are all consistent with a single state f in F, when an arbitrary number of failures may be present.*

4 The Four-Phase Failure Tolerant Protocol

4.1 Roles of clients and servers

A node can play the role of a client or a server in the execution of a transaction. A client generates the request to begin a transaction. A server accepts the request to run a transaction and coordinates execution of the transaction. Every transaction has a defined return value which the server returns to the client.

Nodes concurrently handle multiple tasks. A node acting as a server may also send transaction requests to other nodes, in which case it may simultaneously act as a client. A client may receive requests while it is waiting for a transaction to complete, and act as a server. Some client nodes may be user programs which have a null dataspace, a short lifetime, and only exist so that a user can gain access to the global dataspace of, for instance, a database system. Such a program is considered to be a node in the transaction system and fully participates in the protocol.

When seeking to initiate a transaction, a client contacts a server and sends in a transaction request which specifies what kind of transaction is desired. Clients choose a unique serial number to identify every transaction. This serial number includes the unique name of the client process. One way to do this would be for the client to stamp each request with its unique name, the time of day and a incremental suffix, if more than one transaction request is generated per time tick.

4.2 Eliminating Message Deletions

In the following protocol, a series of messages are sent between processes engaged in a transaction. If a message is lost, and a process is expecting that message, it is assumed that there is a timeout period. For each phase of the protocol, the timeout may be implicitly built into the transaction system, or be specified explicitly by the transaction request itself. If a process is expecting communication from another process, we may now assume that the message is never lost forever, because the receiver will timeout and ask for the message to be resent.

4.3 Four Phases

The are four phases in the processing of a transaction. It is not significant whether or not the nodes participating in a transaction, both client and server, agree on the phase of the transaction at a particular moment in time.

4.3.1 Opening Phase

During the opening phase, a client requests that a transaction be opened. The client sends a server a transaction request which specifies the type of transaction to be performed. The request plus the global dataspace provide all of the information necessary to complete the transaction. For example, if the client node is a user program taking input from a keyboard, there is no way for the user to add any information to a transaction once it is in progress; all contingencies must be specified in advance.

The transaction request includes a unique serial number generated by the client. If the client is a server attempting to open a transaction which is subordinate to some script which it is executing, the serial numbers must be chosen in a special way. When a script generates subordinate transactions, they have the same serial number as the script along with a unique and deterministically generated suffix. (e.g. script #yale-345 might generate three actions, numbered #yale-345-1, #yale-345-2 and #yale-345-3).

The client expects to hear a confirmation from the server that the transaction has started. If a timeout occurs, the client may resend the request using the original serial number.

4.3.2 Commitment Phase

When a server receives a transaction request, it must log the request to ensure that it will be executed, even if failures occur. This is the commitment phase. Every transaction is logged by its server. This includes both scripts and actions, not matter how they are initiated. Even actions which are part of scripts, run on the same nade, are logged.

Once a transaction has been inscribed on the log, it is destined to run to completion, reguardless of how long it takes. Before writing a transaction request on its log, the server checks to see if it has made a commitment to execute any other transaction with the same serial number. It can do this by inspecting its log to see if any transaction request on the log has the same serial number. If so, the request is not relogged. If there is an indication on the log that the request has been completed (closed), its return value is sent back, otherwise, a message is returned saying that the requested transaction is in progress. If the serial number does not match any on the log, the request is logged and a message is returned saying that the transaction is in progress.

Definition 10 *Once a transaction request has been written on the server's log, a transaction has been opened. The transaction remains* open *until it is closed.*

4.3.3 Execution Phase

The transaction enters the execution phase when the server begins accessing the global dataspace. During execution, the server performs the desired transaction. If it is merely an action, the action is run. If it is a script, a series of actions or other scripts may be specified, in which case the server may act as a client and open subordinate transactions. Eventually, the actions needed by the transaction request may be completed (non-terminating scripts are not ruled out). At this time the dataspace will no longer be accessed by the transaction. This is when the transaction enters the closing phase.

Note on nomenclature: A Two-Phase transaction, as widely known, would have two subphases in its execution phase. This is a completely orthogonal issue. The Four Phase Transaction protocol is designed to ensure consistent semantics in the face of failures. Two Phase protocols are designed to ensure serializability.

4.3.4 Closing Phase

During the closing phase, the server sends the client a return value indicating the result of the transaction. For example, the return value could be "5", meaning that the result of the transaction was 5, or it could be "access-denied" meaning that some aspect of the transaction required access to information which was protected. Before sending the return value, the server node writes it on its log along with the serial number of the transaction. If a client becomes tired of waiting for the return value, it may resend the request.

Definition 11 *A transaction is* closed *once its return value has been written on the server's log.*

Upon receiving the return value, the client process acknowledges receipt once it is sure that it will not reexecute the request. Or more formally, a transaction's return value may not be purged from the log until any other transaction which relies on the return value is closed. If transaction t_2, either a script or action, is running subordinate to t_1, the return value from t_2 is not acknowledged, and then purged, until t_1 is closed. Thus, if a node is executing a script, it should wait until the script is closed before acknowledging return values from any subordinate transactions involved in the script. Once a return value is acknowledged, it may be purged from the log of the server node. If a server does not receive acknowledgement after a reasonable amount of time, it may resend the return value and serial number.

A client may receive a return value before it knows that it has submitted a transaction. This could happen if a node began a script, crashed, and then restarted. Subordinate transactions initiated in the first place may run to completion on other nodes and return values. The client would simply throw these values away. As we will see, the client's recovery protocol will set this mess straight. In the meanwhile, the servers for the subordinate transactions must keep the return values on their respective logs, for the client has not acknowledged them yet.

A server node may send acknowledgement of the return value before it is received. This might be done if a node was executing an orderly permanent shutdown. It might not want to wait around for all pending requests to complete. It could acknowledge the closing of all outstanding requests, and shutdown forever.

4.4 Extraordinary Events

There are a few extraordinary events which requires additional consideration by the Four Phase Transaction Protocol.

4.4.1 Crash Failures

Assume that some node crashes without warning. After it restarts it must go through some procedure to ensure that transactions open at the time of the crash are completed consistently.

First, a recovering node must resume state at the previous checkpoint, and then execute in order all actions which are written on its log, whether they are open or closed. Additionally, all open scripts on the log are restarted. The reason that all scripts aren't reexecuted is that scripts can only alter the dataspace via actions, and these script initiated actions are written on the log. If a script is closed, all of the actions which it generated are logged, and it is unnecessary to rerun it.

If there are open scripts on the log, they may not have finished executing before the node crashed; hence, they must be redone The original serial numbers must be reused to ensure that subordinated transactions included in the scripts are also reexecuted using the same serial numbers. In this way, if a script causes a subordinate transaction with another node to be reexecuted, instead of altering that other dataspace a second time, the reexecution will be detected since the serial number is the same. The other node will then send back the return value from the first execution and not actually perform the transaction a second time.

If a script is interrupted by a crash, it will be restarted at the beginning, and we would like for the actions involved in the script to be executed only once. Be keeping the old return values, the actions in the script will only be done once, because they will match up with their original serial numbers. See the lemma in the next section.

4.4.2 Making Checkpoints

Periodically, every node updates its checkpoint structure. This is done by creating a new checkpoint and deleting the old one. Care must be taken to ensure that should a crash occur during this operation, that there is still a valid checkpoint.

When a checkpoint is made, all closed transactions with acknowledged return values are purged from the log. A checkpoint is merely a crash-proof representation of a node's dataspace at a particular point in time. Making a checkpoint allows quick recovery, for otherwise every transaction ever run would have to be stored in the log and re-executed upon restart. The log contains a stream of transactions executed since the last checkpoint.

5 Semantics of the Four Phase Protocol

The Four Phase protocol uses an imperfect distributed system to simulate one which does not have any failures. In the abstract, the protocol's semantics are that transactions are run on a failureless distributed system. Transactions may be executed concurrently by servers, although actions are always serialized on each individual server. Scripts may be interleaved in arbitrary ways, and take arbitrary amounts of time to complete.

In reality, the protocol would constantly be fighting to compensate and correct for message deletions and crashing nodes. We would like to know that the actual semantics of the protocol are equivalent to the abstract semantics which we desire.

Theorem 1 (Failure Tolerance) *The Four Phase Protocol for distributed transaction processing is failure tolerant for crash failures, and for message deletion failures.*

Lemma 1 (Repeatability of Scripts) *If a script's execution is stopped before it is completed, it may be restarted at the beginning, and it will perform exactly the same actions in the same order the second time, with the same serial numbers, that it did the first time up to the point where it stopped.*

Proof of the Repeatability Lemma: The only information available to a script is data from the global dataspace. It can access this data only via actions or subordinate transactions. A transaction with a unique serial number is only executed once. The second time it is attempted, original result is returned, even if redoing the transaction would have yielded a different result.

The return values of subordinate transactions are not acknowledged as complete until the whole script has finished. The return values are stored in the log until acknowledgement is received, and acknowledgement cannot be given by an open transaction. Also, the way serial numbers are assigned ensures that subordinate transactions receive the same same ones on subsequent executions.

The reexecution of any subordinate transaction, be it action or script, will be detected by comparing its serial number to those present on the log, and its initial return value will be sent back without executing it a second time. A reexecuting script therefore receives the same results from its subordinate transactions the second time through. Since these are the orly sources of information to a script, and scripts are deterministic programs, the resulting state must be the same.

The reexecution of a script does not lead to any changes in dataspace, and results in a return to the same point in the initial execution path, after which the script execution proceeds. The only effect of stopping a script in progress and restarting it under the same serial number, is to delay completion of the script. □

Proof of the Failure Tolerance Theorem: It must be shown that the Four Phase Protocol yields consistent results in the presence of an arbitrary number of message deletions and crash failures.

Deletion Failures: By using timeouts, the protocol ensures that a lost message is eventually regenerated. Thus, the only effect of lost messages is to delay the system. Arbitrary delays may occur in an execution, and it will still be guaranteed to be consistent with another execution that has no delays.

Crash Failures: Assume a node N_1 has crashed. N_1 resumes execution with its log intact, by definition of a log, and with an empty dataspace. The dataspace immediately before the crash included the data objects in the previous checkpoint as modified by all actions on the log. In the recovery process, the previous checkpoint is read into the dataspace. All actions on the log are redone, so the former state of the dataspace is restored.

The state of the logfile is not altered during a crash. The checkpoint structure is unaltered. The global clock may advance, but this element of state does not affect the semantics of the transaction system since arbitrary delays are allowed.

No alterations to the log or checkpoint file occur during the recovery process, consequently, a crash during recovery will only restore the node to its state after the first crash. So crashes during recovery may be ignored.

The only other state element of a node are its transactions in progress. If transactions were in progress (open) at the time of the crash, they were written on the log. Some such transactions may have been actions. The execution of the actions on recovery, indeed the execution path of the ongoing transaction stream, remains consistent with an execution without any crashes, if the actions in progress are restarted in the original order, as they are written on the log.

If there are open scripts at the time of the crash, restarting them upon recovery will result in a consistent execution of the transaction stream because the Lemma on Script Repeatability shows that a recovered script execution path is identical to that of the same script in progress before the crash. Consequently, the script returns to the same state it was in immediately prior to the crash. Scripts are allowed to be delayed for arbitrarily long amounts of time. Since a crash or a delay have the same effect, crashes do not alter the semantics of script execution.

Since a crashed node can restore its state to one consistent with its state immediately before the crash, the crash might as well not have happened. All that a crash does is delay the node. Since delays can occur with or without a crash, the presence of crashes does not alter the semantics of the protocol in any way. □

6 Serialization, Synchronization and Deadlock

The transaction system presented here does not serialize transactions or provide an explicit locking mechanism for synchronizing its distributed processes. Locks can easily be established by using a convention. Every lock is represented by an object in dataspace, for example one might be named X. To obtain that lock, a user or script must successfully $delete(X)$. Delete returns a value indicating if the object was successfully deleted. To release the lock, the user or script simply executes $Add(X)$. It takes little imagination to see how an analogous scheme can be created to implement different levels of locking, such as read-only locks, or write locks. The computational model provided by scripts, actions and dataspace is sufficiently rich to support a wide range of algorithms that have traditionally been built into transaction schedulers.

One way to serialize transactions is to use Two Phase transactions. To do this, scripts are carefully designed to meet the following criteria. First of all, every object accessed by a script has an associated lock, and the script always locks the objects it uses. All locks are established in the first subphase of execution. Then locks are released in the second subphase of execution. This is guaranteed to serialize all transactions [1, p.50]. The issue of serializing distributed transactions can be addressed by the way scripts are written, and does not need to be solved by the transaction system's protocols.

As with other serializing systems, deadlock is possible. It is possible to write deadlock avoidance and deadlock recovery algorithms. These must be built into the scripts or user programs; they are not provided by the transaction system. For instance, we could use the convention that all locks must be obtained in lexicographic order. It is known that this will prevent deadlock [6, p. 391].

We see that the Four Phase Protocol has done nothing to solve many difficult problems of database design and transaction processing. It neither increases nor decreases the possibility of serializing transactions or avoiding deadlock. The Four Phase system is just as susceptible to some problems as traditional systems. What is does do quite dramatically, is to separate the problem of recovery semantics from these other issues. It makes it possible to design a distributed system with consistent recovery semantics without worrying about other things. The solutions for serialization and deadlock management can be built on top of the Four Phase Protocol.

7 Directions for Further Work

The system presented here is quite simple and I believe that it can be used to implement a distributed transaction system, such as a distributed database with many servers. Somebody should implement the Four Phase Protocol using an existing database system, and create a truly distributed network database, with multiple servers sharing information, and acting in concept as a single database. A simple relational database engine such as the one described in [5] would be sufficient for this purpose. The difficulties in writing suitable scripts should not be overlooked. How to write scripts which would do interesting and useful things is certainly an area for further research. The existence of a way to create a distributed transaction processing system that is completely resistant to certain kinds of failures is exciting, but it still remains to be seen how to put such a thing to good use. This work may be exceedingly important if large distributed computations are desired. In a large enough system, running for a prolonged period of time, the probability of failures becomes certain. Without failure tolerance built in, it is impossible to complete large distributed computations on real world computers!

8 Acknowledgements

I would like to thank Michael J. Fischer, my thesis advisor, for his enormous assistance in the formation of ideas presented here, and for his help in suggesting many revisions incorporated in this document. Also, I would like to acknowledge the valuable discussions I had with David Gelernter in the early stages of this project which helped motivate the final result.

References

[1] Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, (1987).

[2] Birrell, A.D., Jones, Michael B., Wobber, E.P. A Simple and Efficient Implementation for Small Databases. Digital Systems Research Center, January 1, 1988.

[3] Fischer, M.J., Michael, A. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proc. 1st ACM SIGACT-SIGMOD Symp. on Principals of Database Systems,* pages 70-75. Los Angeles, March, 1982.

[4] Garcia-Molina, H., Salem, K. System M: A Transaction System for Memory Resident Data. Princeton University Technical Report CS-TR-195-88. December 1988.

[5] Hochman, J., Ockerbloom, J. *The Yale Database Guide.* Yale Computer Science Computing Facility, unpublished,(1990).

[6] Korth, H.F., Silberschatz, A. *Database System Concepts.* McGraw-Hill, New York, (1986).

[7] Rothermel, K., Mohan, C.. ARIES/NT A Recovery Method Based on Write-Ahead Logging for Nested Transactions. IBM Research Report J6650, January 18, 1989.