In Proceedings of the 9th Intl. Conf. on Parallel and Distributed Computing Systems (PDCS), pp. 684–689, 1996.

# A Fast and Robust Failure Recovery Scheme for Shared-Nothing Gigabit-Networked Databases

S. Banerjee
Information Science & Telecomm. dept.
University of Pittsburgh
Pittsburgh, PA 15260

P. K. Chrysanthis
Computer Science dept.
University of Pittsburgh
Pittsburgh, PA 15260

**Abstract**

Major technological advances have enabled the development of very high speed networks with data rates of the order of gigabits per second. In the future, wide area gigabit networks will interconnect database servers around the globe creating extremely powerful distributed information systems. In a high speed network, the size of the message is less of a concern than the number of sequential phases of message passing. In a previous paper, we have developed a lock-based concurrency control protocol for *gigabit-networked* databases (GNDB). In this paper, we expand on a log-based recovery protocol that provides efficient recovery in a GNDB with the above mentioned concurrency control scheme.

## 1 Introduction

Several exciting advances are being made in the general area of high speed distributed computing. For instance, the rate at which information can be transmitted [12] and the rate at which information can be processed is increasing. Also, user desktops are being enhanced to the point that servers and clients may be indistinguishable in the future, with regards to computing power and functionality. All of these changes are expected to create very powerful distributed information systems. However, it begs the question whether any changes are required in existing protocols to obtain the anticipated performance increase, and whether existing protocols scale with the size of the new distributed systems. In this paper, a special case of distributed systems, that of distributed database systems is discussed in the above context. We refer to these as *gigabit-networked databases* (GNDB).

Traditional data access and data sharing techniques are not expected to scale to gigabit network rates [6, 11, 10, 14, 2]. Thus if any advantages of a high speed network are to be realized, new schemes are required, that can efficiently utilize the huge bandwidths available. Towards this end, assuming a client-server distributed database system in a shared-nothing environment, in [1], we proposed a lock-based concurrency control protocol, which is a variant of the *strict two-phase locking* [4], but specially tailored for a gigabit wide area environment. This concurrency control scheme reduces the number of rounds of message passing by grouping the lock grants, client-end caching and data migration. However, these are the exact circumstances that make the recovery operation difficult. In this paper, we expand on the skeletal framework of the log-based recovery protocol that was proposed by us in [1]. Distributed concurrency control and recovery algorithms typically require sites to engage in *conversations* (sequential message transfers). The concurrency control and recovery protocols proposed exploit the characteristics of a gigabit network to enhance the performance of the database system, particularly that the size of the message is less of a concern than the number of sequential phases of message passing in high speed networks.

In the next section, we provide background information on high speed networks, and for the sake of completeness, the high speed network specific two-phase locking protocol. The new recovery scheme is presented in section 3, and section 4 concludes the paper.

## 2 Background

We assume a client-server distributed system [3, 15]. Traditionally, the servers have been responsible for maintaining the correctness of the database, and reconstructing the database to a consistent state in case of failures. With the advent of high speed networks, cheaper stable memory and processor power, it is expected that data will be moved between the servers and the clients and both servers and clients will be participating in maintaining their consistency. This means that clients and servers must handle in a coordinated manner the effects of failures and concurrency which are the two basic sources of data inconsistencies. Although the schemes described in this paper are applicable when each client processes multiple transactions concurrently, for the sake of simplicity, we assume that each client processes only one transaction at any given time. This section describes the relevant characteristics of high speed wide area networks (WANs), the client-server distributed database system model, and the concurrency control scheme proposed for a gigabit wide area environment.

### 2.1 Gigabit Network Characteristics

We first discuss the characteristics of the high speed WANs and the traditional low speed networks and understand their differences. High speed WANs differ significantly from the traditional low speed networks. There are two basic components of the delay involved in moving data between two computers: the *transmission time*, i.e., the time to transfer all the data bits, and the *propagation latency*, i.e., the time the first bit takes to arrive. Further, intermediate network components in the path of the data introduce extra delays as well. For instance, when the network is congested, the queueing delays at intermediate switches may be significantly

high. We define the sum of the delays introduced by the intermediate network components and the propagation delay as the network latency. As the data rate in wide area networks continues to increase due to technological breakthroughs, the data transmission delay will decrease almost linearly. However, the signal propagation delay which is a function of the length of the communication link and a physical constant, the speed of light, will remain almost constant, and relative to the data transmission delay, will actually seem to increase. At gigabit rates, the propagation latency is the dominant component of the overall delay [6].

The above basic characteristic of high speed networks (referred to as a high bandwidth-delay product) has significant implications on distributed applications. Moreover, since bits cannot travel faster than the speed of light, and the distance between communicating computers cannot be reduced, the only way to combat propagation latency is to hide it in innovative protocols. This is not to say that the performance of a traditional distributed distributed algorithm will be worse in a high speed environment than in a low speed environment. However, the marginal performance improvement will decrease as the data rate continues to increase. Beyond a certain data rate, there will be no further improvement, no matter what the increase in the data rate is, and unless newer database protocols are developed that are *distance-independent*, scalable performance will not be achieved. This observation has motivated the development of a concurrency control protocol [1], and the corresponding recovery algorithm (presented here), which are the first ones in the family of algorithms which we refer to as *APLODDS* for Algorithms for Propagation Latency Optimization in Distributed Database Systems.

### 2.1.1 Failure Model

The future high speed networking environment will provide quality of service (QoS) guarantees, including high network reliability. Thus, the probability of network partitioning and link failures will be relatively low, and node failures will be the primary consideration. For the purpose of this paper, we do not deal with permanent node failures. Only transient failures are considered (i.e., each node is expected to recover.). Every node may be dynamically classified into two broad types: *reliable* and *unreliable*. The only difference between a reliable and an unreliable node is that if a reliable node fails, recovery from the failure will happen within minutes (due to the presence of a back-up processor, or other fast recovery mechanisms), while an unreliable node may take up to several hours to recover from a failure. In such a situation, two extreme cases of recovery may be considered, depending on the type of node executing the transaction. The server maintains information on the reliability of each node. If there is no information available on a particular node, the server may adopt a pessimistic approach and assume that the node is unreliable. The server is always assumed to be a reliable node.

## 2.2 Concurrency Control in GNDB

A new concurrency control scheme has been developed in [1], that clearly illustrates the effects of the new assumptions. To simplify the discussion, we consider here a distributed database with a single traditional database (DB) server and multiple clients with local processing capabilities. When a client needs a data item, it sends a request to the DB server which responds with the requested data item. Let us assume that each client executes one transaction at a time. In the presence of concurrent requests from different clients, the DB server ensures data consistency by following the well known *strict two-phase locking* concurrency control protocol (2PL) [4]. A transaction can access a data item only if no other transaction has a lock on it. In phase 1, a transaction requests data items which are shipped to it after the server acquires a lock on them. In phase 2, all the locks are released when the transaction is committed and all modified data items are returned to the server.

Assuming that a transaction needs to access $n$ data items, the first phase of the 2PL protocol as described above will involve $n$ requests from the client to the server and $n$ replies from the server to the client, exchanged in minimum 2 messages if all requests are sent at the same time or maximum $2n$ messages. The second phase of 2PL will involve a single message. That is, for each transaction, in the best case, strict 2PL involves three *rounds*, i.e., sequential phases of message passing corresponding to lock request, lock grant and lock release.

One of the motivations in a high speed environment is to minimize both the number of messages as well as the rounds. The following scheme proposes to reduce the number of phases of message passing by *grouping* the lock (data) granting and release. The DB server collects the lock requests for each data item for a specified interval. At the end of this interval (referred to as the *collection window* from now on), the lock is granted to the first transaction, and the data item is sent to the respective client along with the ordered list (also referred to as the *forward list*) of the clients that have pending lock requests for that data item, that arrived within the window. Within each window, the forward list may be created according to one of several rules (See [1] for details) to improve performance further. While the data items have been sent out to a group of clients, the server continues to collect requests.

When a transaction commits, the client sends the new version of the data items to the clients next on the respective forward lists. A copy of the forward list is also sent with each data item. If the transaction aborts, the client forwards the unchanged data to the next client. Finally, when the last client on the forward list terminates, it sends the new version of the data to the server with the outcome of each transaction executed on the clients on the forward list.

In this scheme, the lock release message of the previous client is combined with the lock grant message of the next client, thereby eliminating one sequential message in the protocol. For example, assume $n$ clients and a single data item. The 2PL scheme will require $3n$ messages and $3n$ rounds as opposed to the proposed scheme which will require $2n + 1$ messages and $2n + 1$ rounds. Clearly, the messages in the proposed scheme have a larger size than that in the 2PL scheme, but in a gigabit environment, the size of a message is not a big consideration. Note that this group granting and release of locks is not possible when the DB server alone is responsible for processing the data.

While not explicitly discussed here, the concurrency control scheme in [1] can handle the case of shared access as well. A detailed simulation study of the performance of this concurrency control scheme is underway. However, a few comments are in order here. While no data access patterns have been assumed, note that the more a certain data item is requested such as hot data items, more is the performance gain, since the grouping effect is more emphasized when the forward list is longer. Also, to keep the paper more focused, several enhancements to the basic scheme have not been discussed.

## 3 Failure Recovery in GNDB

Although the recovery scheme described here is in the context of the concurrency control protocol described in the previous section, it can be applied in systems that support data migration or data shipping in a shared-nothing environment. The recovery scheme described in [8] supported data shipping in a shared-disk environment. Note that the recovery scheme in shared-nothing systems that support data migration is quite difficult, as supported by previous work in [13, 2]. The scheme described here *not* only achieves the objective of failure resiliency, but is also efficient in the number of message passing rounds, as warranted in a high speed network environment. We do not currently discuss granularity issues, and assume that a data item is the unit of concurrency control and recovery as well as the unit of migration. Thus our scheme avoids data dependencies induced by data migration and supports *isolated failure atomicity* [9] ensuring that transactions executing on operational nodes are not affected by crashed nodes. The physical size of a data item may range from the record level to several pages long.

Most database systems maintain some sort of recovery information which can be used to recover the data in case of failures. The type of recovery information maintained depends on whether the system has *in-place* updating or *out-of-place* updating [5]. When updates are performed in-place, the value of a data item is directly changed by a transaction updating that data item, thus losing the previous values of the data item. Under such a scenario, information about the operations of each transaction are maintained in a separate repository called the database log. In case of a failure, it is possible to either REDO (if the transaction is committed) or UNDO (if the transaction is aborted) any operation of a transaction, as the case may be. On the other hand, when updates are performed out-of-place, that is, the old data is left intact, and the updated information is copied to another memory location. The new copy is called the shadow copy. The old value of the data item is retained for recovery (UNDO) operations. Shadowing and logging may be used together as well. In-place updating, and hence log-based recovery is more common than out-of-place updating or the shadowing approach. A log-based recovery scheme for a GNDB is proposed next.

Recovery is very difficult in a situation where data items may migrate from site to site [13]. The new recovery scheme proposed here is designed to work in conjunction with a concurrency control scheme involving data migration (like in [1]). In most client-server configurations with multiple servers, the servers are responsible for the recovery operation, and achieve this with a commit procedure (typically, the two-phase-commit (2PC) scheme.). However, in a gigabit environment, one of the motivations is to reduce the message passing rounds. The 2PC mechanism requires 3 rounds of message passing per transaction between a coordinator site (typically, the site that initiates the transaction) and several participant sites (the sites that participate in the processing of the transaction). The 3 rounds consist of the *prepare-to-commit* messages from the coordinator to the participants, a *decision* message (abort or commit) from the participants to the coordinator and then the *commit* or *abort* messages from the coordinator to the participants. Even if one participant server replies with an abort message, the transaction is aborted. On the other hand, to commit a transaction, all participants have to agree to commit.

In the concurrency control mechanism in [1], data items migrate from client to client in each window, and each transaction can complete execution when all the data items required are available at that client, a very simple and fast commit procedure can be adopted, i.e., every transaction in a window can commit locally. While the server is responsible for the recovery of the database, the clients record each modification to a data item in a log on stable storage and pass around the corresponding log records along with the data item. The client discards a log entry when the log entry is stored on the server's log. This would reduce the number of rounds of message passing incurred in the commit procedure to zero. While this is extremely desirable in a gigabit environment, some other crucial problems arise. Before getting into the details of the recovery process itself, it is not hard to see that while a data item is granted to a group of clients, a server cannot recover the data item when a client fails until the failed client recovers. Thus, in a failure-prone environment a more efficient recovery scheme is required. If the site that fails is reliable, obviously, the penalties are not as severe as when an unreliable site fails.

A more cautious recovery scheme would require that the server be informed about the outcome of each transaction and its associated log records as soon as possible. This can be achieved by requiring each client to send to the server the new version of the modified data at the same time when the new version is forwarded to the client next on the forward list. Note that the server needs only to be informed for the modified data. Although, the new resilient scheme requires maximum $3n$ messages, same as the 2PL scheme, it requires only $2n+1$ rounds, as opposed to $3n$ of the 2PL scheme. An even more efficient recovery scheme might be possible at the cost of more messages with the advantage of less rounds. In the following, we develop such a recovery scheme that adapts to the level of reliability of a site and reduces the overall cost of recovery in a GNDB.

### 3.1 An Adaptive Recovery Scheme

We first discuss the recovery operation for reliable sites, and then follow up with the more interesting case of unreliable sites. Depending on the access sets of transactions in a collection window, a transaction precedence graph may be created. The transaction precedence graph is a directed graph which determines the order in which each data item will *move* from one client site to another. Each transaction that immediately precedes a transaction in a precedence graph for

Figure 1: Example depicting multiple predecessors and successors of a transaction

a data item, is termed a *predecessor* transaction, and a transaction that immediately follows is termed a *successor* transaction[1]. The last site in each precedence graph is always the server, so the last client(s) can return the data item to the server, which then serves the next window. Obviously, a transaction may have multiple successors and predecessors and the set of successors/predecessors must be determined from the precedence graphs of all data items accessed by the transaction. The set of all successors (predecessors) for a transaction is termed as a successor (predecessor) set. For example, consider the case of a transaction at a client C that requires exclusive access to four data items *W, X, Y* and *Z*. In Figure 1, a case is depicted where site C has three predecessors $P_1$, $P_2$ and $P_3$, and four successors $S_1$, $S_2$, $S_3$ and $S_4$. The arrows indicate the direction of data transfer. For data item $W$, site C is a successor to site $P_1$, and site $S_1$ succeeds site C for the same data item. Thus, site C is an immediate successor of site $P_1$, and the successor of the successor of site $P_1$ is site $S_1$. Several cases need to be considered in the development of the recovery protocol depending on the whether the site processing the transaction, its predecessors and successors are reliable or unreliable. The following possibilities exist: (1) The site processing the transaction may be reliable or unreliable, (2) The predecessors may be reliable or unreliable, and (3) The successors may be reliable or unreliable.

*Recovery Operations for the Servers*

Each server in the system is assumed to be reliable, implying that after a crash, it will recover relatively soon. However, for a server to resume normal operation after a crash, it needs the following two vital pieces of information: (1) Information about transactions that had committed before the crash, and whose effects have not yet been made permanent in the database, and (2) Information on the contents of the forward lists for each data item.

Since clients commit their transactions locally, and then inform the servers, there are no UNDO operations at the server. During the recovery process after a crash, a server will reconstruct the database into a state consistent with all the committed transactions known to the server using REDO operations. Thus, on receiving a commit message from a client, the server needs to log the commit operations in stable storage before acknowledging the commit message to the client. Further, after each collection window duration, before a data item is sent out to the first client on the forward list, the forward list as well as the data item are force written to stable storage. This ensures that after a crash, the server will obtain the above two pieces of information, and resume normal operation.

*Recovery Operations for the Clients*

The recovery operations for clients depends on the reliability of the successors of the transaction. To maximize the availability of data items, the following commit procedures may be adopted for transactions at reliable and unreliable sites. All clients are assumed to support stable storage facilities. Further, the servers now send extra information with the forward lists, about the reliability of each client site on each forward list.

**A. Reliable Sites:** Reliable sites are expected to be able to recover from failures relatively quickly, and hence the commit procedure is less complex. Reliable sites are assumed to support stable storage, combined with an efficient write-ahead-logging (WAL) scheme, e.g., Aries [7]. When a reliable site receives a data item, the site force-writes the data item along with its forward list to the stable storage, and then sends an acknowledgment message to its predecessor. Once the site receives all the data items required by its transaction, it finishes executing the transaction, making appropriate log entries using the efficient write-ahead-logging (WAL) scheme. Then, it can commit locally and send the data items and the log entries to the appropriate successors on the forward list. The data items and log records are preserved until all the sites that receive the data and log acknowledge the receipt of the data and the log records. After that, the cache at the site can be discarded (garbage collection).

If the successor site for a data item is unreliable, a copy of the data item and the log records is sent to the successor of the unreliable site as well for that data item. If the successor of an unreliable successor happens to be unreliable, then a copy of the data item and the log records is sent to the server as well. The reason for this will become clear when the recovery process for an unreliable site is discussed. If a reliable client fails in the middle of executing a transaction, during the recovery process, UNDO (if the transaction is aborted) or REDO (if the transaction is committed) operations as the case may be, are executed to bring the cached data items to their consistent state. While the site is in the failed state, the data items cached at that site will be unavailable. Using the above scheme, transactions can be committed very quickly without any extra rounds of communication with the servers or other clients.

**B. Unreliable Sites:** The recovery mechanism used for unreliable sites is more complicated, and involves communication with other sites. In the event of a site failure, the objective here is to avoid blocking the operation of all the other transactions that require the data items after the transaction at the unreliable site. Note that each successor site may require only a subset of the data items held by the transaction at the failed site. If the site processing a transaction fails, there needs to be a method of *bypassing* the failed site, so that the successor sites can continue operation, either with the after-

[1] We refer to the site at which a predecessor transaction is processed as a predecessor site, and similarly the site at which a successor transaction is executed is termed a successor site.

images of modified data items, or the before-images of unmodified data. The main problem stems from the need to ensure that every successor site of a transaction comes to the same decision regarding the transaction, viz., the transaction is committed or aborted. Next, we propose an atomic commit protocol that allows the set of successors to reach a consistent decision, and gain access to the correct data.

As mentioned before, when the successor of a site (reliable or unreliable) for a data item is unreliable, the data, the logs and the forward lists are sent to the successor of the successor as well. Thus, each successor of an unreliable site obtains the before-images of the data items required by it, as well as learns the identity of their unreliable predecessors. The successors of an unreliable site monitor it periodically to see if the site has failed. If the unreliable site fails, then its successors consult each other and vote to abort or commit the transaction at the unreliable site. If the transaction is voted to have aborted, then the before images of the data are used by the successors. Else, the after images are used in any subsequent transactions. The voting procedure is described later.

When the transaction at the unreliable site receives all the data items required (along with the respective forward lists and logs), it constructs a list of all its successors from the forward lists. If a site is concurrently processing $n$ transactions, it will be part of $n$ successor site sets. Here it is assumed that $n = 1$. This list is then sent to all the members of the successor set. The members of the successor set store the identities of the successor set in to stable storage before sending an acknowledgment. When the transaction at the unreliable site is ready to commit, it writes a "Ready to commit" entry into the stable log, then sends the after-images of *all* the data items and the logs to the set of its successors. If all its successors happen to be unreliable, then a copy of the data and the logs is also sent to the server. Under this circumstance, the server is considered to be part of the successor set, and this information is given to all the successors. Again, as in the reliable site case, if the successor for any data item happens to be unreliable, that data item and associated log entries are sent to the successor of the successor. Once the after-images are broadcast to the set of successors, the transaction will wait for at least one acknowledgment, and will repeatedly try to elicit a response from the successor sites (and the server, if all its successors are unreliable.) in case it does not receive the acknowledgment within a specified time-out period. The acknowledgment serves as only a guarantee that at least one of the successor sites (or the server, if applicable) has the after-images. The transaction is committed (a "Commit" log entry is made) only after it receives the first acknowledgment. The first acknowledgment will typically arrive from the physically closest, and/or the most lightly loaded site at that time. When all the acknowledgments have been received, the site may discard all information on the transaction just executed (garbage collection).

If the unreliable client site fails before or after sending the after-images, the successor sites that do not receive the after-images within a specific time-out, initiate a voting process to abort the transaction. The votes are requested from the other members of the successor set. Even if one member of the successor set has received the after-images, it sends the after-images to all the successors, which can then proceed with their respective transactions. The voting scheme called the two-phase abort (2PA) is described in detail below.

*The 2PA Voting Mechanism*

In the first step of the voting phase, the vote initiator (which may be one or more successor sites that time out after not receiving the after images) sends a vote-request to abort the transaction at the unreliable site to all the successors (and the server if applicable) of the failed site. Even if one successor site has received the after-images, it will send a negative vote along with the after-images to all the successors (and the server, if applicable). Thus the successors can proceed with the execution of their respective transactions. If none of the successors (or the server) have the after-images, the predecessor transaction may be assumed to have aborted, and the before-images of the data. All successors have to vote yes to abort for a transaction to be aborted, and hence the name 2PA.

When the failed site recovers, and sends the after-images to the successors (and the server if applicable), abort messages will be returned to the recovered site if the successors have voted to abort the transaction. The site will then have to abort the transaction. If the transaction was voted to be committed, then the successors will send acknowledgments and the transaction will be committed. Thus, even under the very improbable circumstance that *all* the messages containing the after-images are lost, the successors to a transaction will be able to proceed on the assumption that the predecessor transaction was aborted. It should be noted that the new scheme is being proposed for a high speed environment with QoS guarantees, and hence it is indeed extremely unlikely that all the messages will be lost. In fact, with emerging high speed networking technology, it will be possible for applications to *demand* a particular grade of service. Thus, for recovery mechanisms, it will be possible to specify to the network that no message loss will be tolerated. Note that with the scheme outlined above, clients will be able to perform *partial rollbacks* without involving the server, since all data items required by the client are cached at the client before it can commit. The complete recovery algorithm including the voting protocol for unreliable sites is specified in Figure 2, with a detailed example.

**Example:** Consider the system in Figure 1 and the following representative cases. For simplicity's sake, assume that all 4 data items are managed by a single server.

Case 1: Client sites C, $P_1 - P_3$ and $S_1 - S_4$ are reliable.

Since all the predecessors of C are reliable, each of them commits locally, and then transmits the 4 data items to C. Since C is reliable, copies of the data are forwarded only to C, and not to its successors. C copies the data items and the forward lists to stable storage before acknowledging each predecessor. The transaction at C is executed with appropriate WAL operations to stable storage. If C fails at any time, the 4 data items will be unavailable. On recovery, C will do the appropriate REDO/UNDO operations to reconstruct its local cache to a consistent state. Once the transaction is completed, C will transmit the 4 data items, the respective logs and forward lists to the appropriate successors (W to $S_1$, Z to $S_2$, X to $S_3$, and Y to $S_4$). Since all the successors are reliable, the data will not be forwarded to any other sites. The successors will cache the data in

stable storage and then send acknowledgment messages to C. Once C has received all four acknowledgments, it will execute garbage collection and remove the logs pertaining to the transaction from its stable cache.

Case 2: Client sites C, $S_1$ and $S_4$ are unreliable.

In this case, when the three predecessors of client site C commit their transactions, they send the 4 data items along with the logs and the forward lists to C. However, since C is unreliable, all the successor sites of C, for each data item, are sent the data, logs and forward lists as well. For instance, successor $S_1$ receives the data item W along with its associated information, successor $S_2$ receives data item Z, and so on. On receipt of the 4 data items, C constructs the list of its successors from the 4 forward lists, and sends each of them a copy of the other 3 data items, logs and forward lists as well as the identities of the other successors. The successors cache this information in stable storage before acknowledging C. At this point in time, each successor has a copy of all the information that C has. Since there is at least one reliable site in the successor set, the server is not involved. C continues to execute its transaction, and making appropriate log entries using WAL.

When the transaction is ready to be committed, C sends a copy of the after images of each of the 4 data items to all its successors. C is allowed to commit the transaction only after it gets at least one acknowledgment (typically from the site that is the closest and/or the most lightly loaded) from a successor[2].

If C fails at any time, the successors monitoring C will time out, and using the successor list sent before, will solicit a vote-to-abort from the other successors. Even if one successor has received the after-images, and votes to commit, the other successors will use the after-images to continue the transaction. However, to abort the transaction, all successors will have to vote to abort (and hence the name 2PA). When C recovers, resumes operation, and tries to commit its transaction, it will receive an acknowledgment from a successor only if the voting process resulted in a commit decision. Else, the successors will return negative acknowledgments, and C will have to abort its transaction. If C fails before providing the list of successors to all the successors, the successors will have to consult the server for this information.

Case 3: Clients C, $P_1 - P_3$, $S_1 - S_4$ are unreliable.

This case is exactly the same as case 2, except that since all the successors are unreliable, the server will be included in the successors list. Thus in the very remote possibility that all successors have failed, the server can intervene so that subsequent transactions are not blocked.

## 4 Conclusions

In this paper, a recovery procedure in the family of algorithms termed as APPLODS (Algorithms for Propagation Latency Optimization in Distributed Database

[2]Note that if the acknowledgment comes from an unreliable successor, and this successor fails after acknowledging C, during the voting process, it will not be able to answer and hence subsequent transactions will be blocked. We can overcome this problem by requiring an acknowledgment from a reliable successor before allowing C to commit. However, this may require a larger network latency. This decision will need to be made using the actual reliability parameters of the system.

- If the successor is unreliable, send committed data, logs and forward lists to the next two successors on each forward list.
- Once all the data items are received by an unreliable site, it constructs its successor set from the forward lists, and sends this information to all the members of the successor set.
- If waiting for a release from the immediate predecessor (after having received a message from the pre-predecessor site), after a timeout, the predecessor site (pred-ID) is declared to be failed,
  - Initiate the 2PA voting procedure with the successors of pred-ID. If all successors of pred-ID are unreliable, include the server in the voting process.
    **The 2PA Voting scheme**
    - Phase 1: Vote initiator sends to all successors of pred-ID a request-to-abort message, along with the previous uncommitted data, and the list of all predecessors and successors of pred-ID.
      - If a successor has received committed data (and acknowledged) from pred-ID, it broadcasts the data (and the corresponding forward lists) to all the successors of pred-ID.
      - Otherwise, it sends its yes vote-to-abort to the successors of pred-ID.
    - Phase 2: All successors of pred-ID decide to abort the message after getting *all* yes-votes from the successors of pred-ID.
- On receiving a vote-to-abort message,
  - if a successor has received after-images of a data item from pred-ID,
    - it will broadcast the released data to all the successors of pred-ID.
  - else, it will vote yes-to-abort pred-ID.
- When a successor receives a message from either the server or one of the successors with the committed data from pred-ID, it resumes normal operation with the new data.
- When a predecessor of pred-ID receives the abort messages from all the successors and the server, it removes pred-ID from the relevant forward lists.
- If the predecessor of pred-ID has already forwarded the data to pred-ID, it ignores the abort messages.
- If any successor or the server receives the committed data from pred-ID after a vote-to-abort has been passed, it sends an abort message to pred-ID.
- After pred-ID recovers, it tries to elicit acknowledgments from at least one of the successors or the server. If it receives an abort message in response, it aborts its transaction.

Figure 2: Unreliable Site Recovery Protocol

Systems) that are suited to the gigabit WAN environment has been proposed. Previously, a concurrency control scheme suited for the high speed environment was proposed by the authors. However, in order to reduce the effects of network latencies, this scheme required data migration, where the recovery process is typically very difficult. The major contribution of this paper is the development of a practical recovery process in conjunction with the previously proposed data sharing scheme. This adaptive recovery scheme distinguishes between reliable and unreliable sites and proposes a different recovery procedure in each case. With reliable sites, the recovery process is simple, and does not incur any extra rounds of message passing that what is required for the concurrency control protocol. The problem with unreliable sites is that if they fail while executing a transaction, any transactions that require the same data items will be blocked. The recovery procedure proposed for unreliable sites circumvents this problem, by a voting mechanism by the successors of the unreliable site. This new voting scheme called the two-phase abort (2PA) is a byproduct of this research. Work is currently in progress to develop a simulation model of the concurrency control and recovery schemes in the APPLODS family of algorithms to demonstrate the expected improvement in performance. One of the crucial problems that remains to be solved is the accurate estimation of the time-out periods before a predecessor is declared failed by the successors. The simulation model will provide insight in to this estimation problem. We also plan on extending the proposed schemes to other configurations of client-servers.

## References


[1] S. Banerjee and P. K. Chrysanthis. Data Sharing and Recovery in Gigabit-Networked Databases. In *Proc. of the Fourth Intl. Conf. on Computer Comm. and Networks: IC3N-95*, pages 204–211, 1995.

[2] S. Banerjee, V. O. K. Li, and C. Wang. Distributed Database Systems in High-Speed Wide-Area Networks. *IEEE Journal on Selected Areas in Comm.*, 11(4):617–630, 1993.

[3] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *Proc. of the ACM SIGMOD Conf.*, pages 357–366, 1991.

[4] K. P. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Comm. of the ACM*, 19(11):624–633, 1976.

[5] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[6] L. Kleinrock. The Latency/Bandwidth Tradeoff in Gigabit Networks. *IEEE Comm. Magazine*, 30(4):36–40, 1992.

[7] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[8] C. Mohan and I. Narang. Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment. In *Proc. of the 17th VLDB Conf.*, 1991.

[9] L. Molesky and K. Ramamritham. Recovery Protocols for Shared Memory Database Systems. In *Proc. of the ACM SIGMOD Conf.*, pages 11–22, 1995.

[10] C. Partridge. *Gigabit Networking*. Professional Computing. Addison-Wesley, 1993.

[11] C. Partridge. Protocols for High Speed Networks: Some questions and a few answers. *Computer Networks and ISDN Systems*, 25:1019–1028, 1993.

[12] R. Ramaswami. Multiwavelength Lightwave Networks for Computer Communication. *IEEE Comm. Magazine*, 31(2):78–88, 1993.

[13] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A New Architecture for Distributed Data. In *Proc. of the Tenth Intl. Conf. on Data Engineering*, pages 54–65, 1994.

[14] J. D. Touch and D. J. Farber. The Effect of Latency on Protocols. In *Proc. of the ACM SIGCOMM*, 1994.

[15] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/server DBMS Architecture. In *Proc. of the ACM SIGMOD Conf.*, pages 367–376, 1991.