Data Sharing and Recovery in Gigabit-Networked Databases

Sujata Banerjee
Telecommunications Program
University of Pittsburgh
Pittsburgh, PA 15260

*Panos K. Chrysanthis**
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Major advances in optical fiber transmission and switching technology have enabled the development of very high speed networks with data rates of the order of gigabits per second. It is anticipated that in the future, wide area gigabit networks will interconnect database servers around the globe creating extremely powerful distributed information systems. In this paper, we examine the implications of such a high speed network on data access and sharing techniques and propose a lock-based concurrency control protocol and a log-based recovery protocol that ensures data consistency in gigabit-networked *databases. Both protocols exploit the characteristics of a gigabit network to enhance the performance of the database system and, in particular, the fact that the size of the message is less of a concern than the number of sequential phases of message passing.*

1 Introduction

The evolution of very high speed networks is prompting research in many areas, including that of distributed database systems of the future. These networks will have speeds of the order of Gigabits per second, and may even increase to Terabits per second someday [1]. It is anticipated that in the future, wide area gigabit networks will interconnect database servers to clients around the globe creating extremely powerful distributed information systems. We refer to these as *gigabit-networked databases* (GNDB). A good example of such a system is the proposed National Information Infrastructure, which is expected to provide fast, and reliable access to correct diverse data. Traditional data access and data sharing techniques are not expected to scale to gigabit network rates [2–7]. 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.

At gigabit speeds, migrating large amounts of data from the database servers to the clients (and vice versa) will not pose a problem (Figure 1). Furthermore, in the future clients will be equipped with specialized hardware and execute specialized computations not supported by traditional database servers. Thus, high speed networks will significantly change the traditional client-server operating environment [8, 9], where typically the servers do most of the processing. In this new operating environment, 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 concurrency and failures which are the two basic sources of data inconsistencies.

In this paper, we propose a lock-based concurrency control protocol, a variant of the *strict two-phase locking* [10], and a log-based recovery protocol [11] that ensures reliability in such a client-server database environment. Distributed concurrency control and recovery algorithms typically require sites to engage in *conversations* (sequential message transfers involving round-trip propagation delays). Both of the proposed protocols exploit the characteristics of a gigabit network to enhance the performance of the database system, particularly the fact 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 elaborate on these characteristics of high speed networks (Section 2). Then, in Section 3 we present the high speed network specific two-phase locking protocol whereas the recovery scheme is presented in Section 4. Section 5 concludes the paper with a summary and a discussion on future steps.

2 Background

Before introducing our high speed network-specific concurrency and recovery protocols, it is important to first discuss the characteristics of the high speed wide area networks (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[1] 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. As the data rate in wide area networks continues to increase due to technological breakthroughs in optical fiber transmission and switching techniques, 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

*This research is supported in part by the National Science Foundation under grant No. IRI-9210588.

[1] Queuing delay at intermediate switches is ignored.

0-8186-7180-7/95 $04.00 © 1995 IEEE

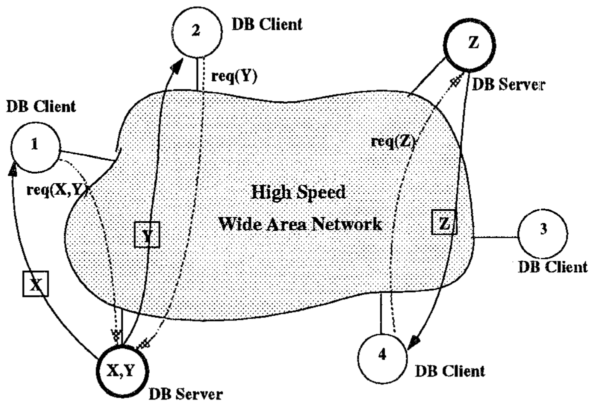

Figure 1: Distributed Database System connected by a High Speed WAN

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 [2]. For example, the propagation delay across the United States (at the speed of light) is 20 milliseconds. At a network speed of 1 Mbps, the transmission delay for a 1 Mb file is 1 second, and the ratio of the propagation delay to the transmission delay is 0.02. At 1 Gbps, the same ratio is computed to be 20, a 1000 fold increase over the previous value. Most existing protocols do not exhibit scalable performance over such a wide range of variation of this ratio [12]. Thus, the problem of propagation latencies actually gets worse as the data rate increases.

This basic characteristic of high speed networks (also 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 algorithm will be worse in a high speed environment than in a low speed environment. Due to the lower data transmission delays in a high speed network, the protocol will perform better in a high speed network, but 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 the two algorithms proposed 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.

3 Concurrency Control in GNDB

With the above issues in mind, a new scheme has been developed 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 requirements. When a client needs a data item, it sends a request to the DB server which responds with the requested data item. Let us also assume that a client executes one transaction at a time. In the presence of concurrent requests from different clients, the DB server preserves data consistency by following the *strict two-phase locking* protocol (2PL) [10], the most commonly used concurrency control mechanism. The 2PL protocol ensures data consistency as defined by *serializability* which requires the concurrent, interleaved, execution of requests to be equivalent to some serial, non-interleaved, execution of the same requests [11].

The 2PL protocol executes each transaction in two phases. A transaction can access a data item only if no other transaction has a lock on it[2]. During the first phase, a transaction requests data items which are shipped to it after the server acquires a lock on them. In the second phase, 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 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. The time for each round may vary according to the distance between the server and the client, the client loading, message route taken, etc. However, here we focus on the propagation latency, and hence the distance between a particular client and the server.

As mentioned before, 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 time interval. At the end of this interval (referred to as the *collection window*), 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 that arrived within the window. As discussed below, within each window, the forward list may be created based on several rules to improve performance further.

When a transaction commits, the client sends the new version of the data item to the client next on the forward list along with the forward list. If the transaction aborts, the client forwards the version of the data that it has received to the next client. Finally,

[2]Concurrency can be enhanced by distinguishing between shared (read) and exclusive locks. Several transactions can access a data item simultaneously using a shared lock. To keep the current discussion simple, only exclusive locking is assumed.

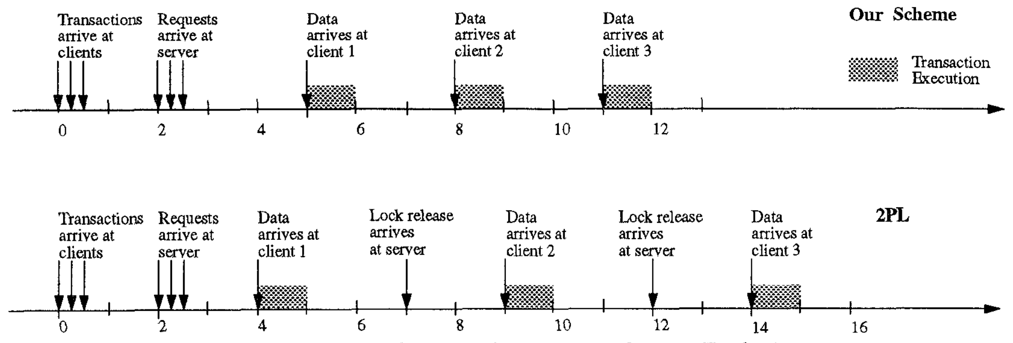

Figure 2: Example execution of our scheme: Exclusive access

when the last client on the forward list terminates, it sends the new version of the data to the server along with the forward list which also reflects the outcome of each transaction executed on the clients.

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 required by the 2PL protocol between a client and the server. For example, assume that n requests for the same data item arrive within the same window. 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. Note that this group granting and release of locks is not possible when the DB server does all the data processing. The following example demonstrates the working of this basic scheme.

Example: Consider a system with one DB server, and three clients numbered 1–3. Assume each client has issued a transaction (say, T_1, T_2, and T_3) that exclusively access the same data item. Assuming that each message/data transfer is accompanied with 2 units of propagation latency. Since a high speed networking environment is assumed, the message/data transmission time will be negligible. Let the collection window duration be 1 unit, and the processing time per transaction after receiving the data item be 1 unit. The collection window starts when the first transaction arrives. It is also assumed that all three transactions arrive within the same collection window. Figure 2 depicts the execution for the new protocol and compares it with 2PL. The total execution time for our scheme is 12 units, versus 15 units with 2PL.

While the gains from the new technique may seem modest from the above example, under higher transaction rates, as the queues build up, it is possible to demonstrate a significant performance improvement. The rest of this section expands on the basic scheme.

3.1 Allowing Shared Access

In the above description of the basic scheme, only exclusive access to data was considered. Obviously, access to some data may be done in a shared fashion, with multiple clients *reading* the data item simultaneously. Thus, shared access needs to be incorporated into the basic scheme. However, in the interest of strict consistency, while multiple clients may read the data simultaneously, no client may write on it until the clients reading the data have released the shared lock. Actually, as it will become evident below, we can do better than this by allowing multiple readers and a single writer to execute concurrently while still preserving strict consistency.

For each data item required in the shared mode by multiple (reading) clients, the DB server can send a copy of the data item to each of the reading clients, with the forward list containing the client C_i that requires the data item next in the exclusive mode. At the same time, a message containing the data item and the list of the shared-mode clients is also sent to C_i that requires exclusive access. Although this enables C_i to execute concurrently with the reading clients, C_i cannot release its updates until it receives a *release* message from all the reading clients. Here is interesting to point out that the protocol just described behaves similar to the two-copy version 2PL protocol [11] which allows more concurrency than the standard 2PL protocol. As before, if there are no waiting transactions that need exclusive access, the release messages are returned to the server. If there are n clients reading a single data item, $3n$ messages in 3 rounds will be required.

Example: The timing diagram for another example is shown in Figure 3, where transactions T_1 and T_2 require shared access and T_3 requires exclusive access to the same data item. The total execution time for our scheme is 9 units, while with 2PL, it takes a little more than 10 units. The figure does not depict the best case in which the lock is released by client 3 one time unit earlier, that is, at the time immediately when it receives the release message from clients 1 and 2.

If all three transactions required shared access, then the 2PL scheme would have required a little more than 5 units of time, while our scheme would have required one more unit of time than the 2PL scheme. The performance can be improved by reducing the collection window appropriately (in this case by 1 unit). Thus, collection window duration needs to be tied to the statistics of shared and exclusive access in the system.

In order to illustrate the behavior of the protocol when more than one data item is requested by each

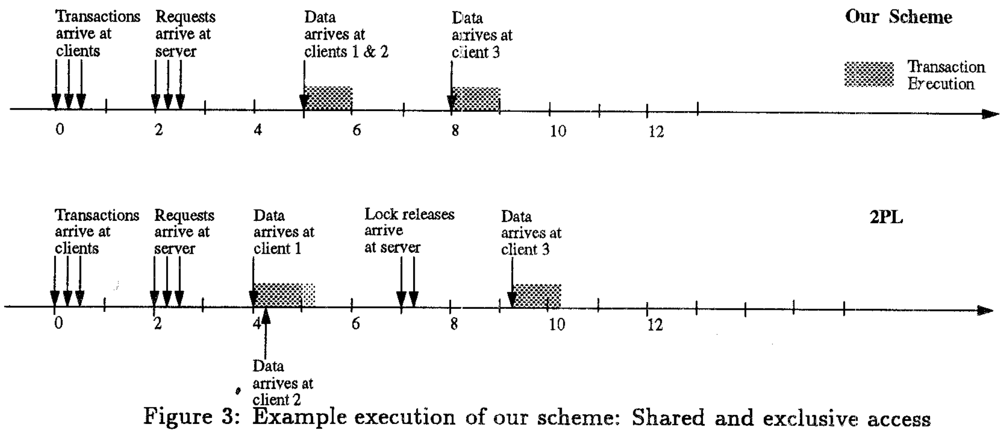
Transactions arrive at clients
Requests arrive at server
Data arrives at clients 1 & 2
Data arrives at client 3
Our Scheme
Transaction Execution
0
2
4
6
8
10
12
Transactions arrive at clients
Requests arrive at server
Data arrives at client 1
Lock releases arrive at server
Data arrives at client 3
2PL
0
2
4
6
8
10
12
Data arrives at client 2

Figure 3: Example execution of our scheme: Shared and exclusive access

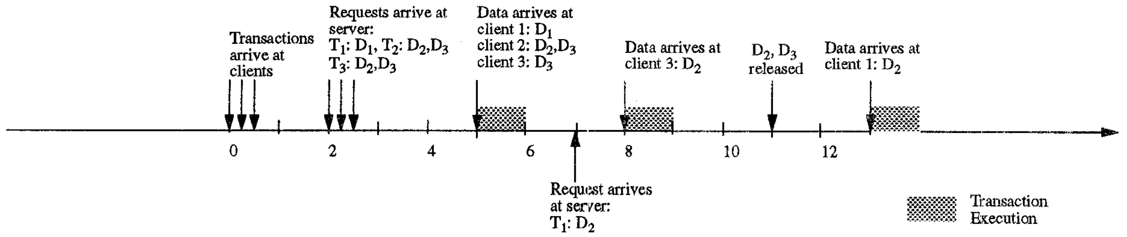
Transactions arrive at clients
Requests arrive at server: T1: D1, T2: D2,D3 T3: D2,D3
Data arrives at client 1: D1 client 2: D2,D3 client 3: D3
Data arrives at client 3: D2
D2, D3 released
Data arrives at client 1: D2
0
2
4
6
8
10
12
Request arrives at server: T1: D2
Transaction Execution

Figure 4: Example execution of our scheme: Dynamic shared and exclusive access

transaction, let us consider an example with three transactions as before, except that now each transaction requires two data items, as given below. For each data item, the type of access (Exclusive: e, or shared: s) is denoted as the superscript. $T_1 : \{D_1^e, D_2^e\}$, $T_2 : \{D_2^e, D_3^s\}$, $T_1 : \{D_2^e, D_3^s\}$

Let us assume that all the requests for data items arrive within the same window, except for T_1 which requests D_2 after some initial processing. The timeline for this scenario is provided in Figure 4.

In this example, the server can wait until the data D_2 is released before sending it to client 1. However, imagine the following scenario, where T_1 requests D_2 at a later time, while T_2 requests D_1 at a later time. Under such circumstances, the server detects a deadlock, and has to abort one of the transactions (preferably the one with the least number of locks, or the one that has spent the least amount of time in the system). In the next section, we will discuss how careful construction of the forward list potentially reduces the number of deadlocks. In general, we assume that the server detects deadlocks by maintaining a *wait-for* graph and checking for cycles in the graph [11].

3.2 Creating the Forward List

For each data item, in each window, a forward list is created during the time period that the requests from the previous window are being served. This is basically performed in two steps. These two steps can be performed sequentially as described below, or in an interactive manner, during which the forward list is built incrementally.

First Step: In the first step, the forward lists are constructed based on some ordering rules. There are many ordering rules possible for each forwarding list, with different performance implications:

- First-in-First-Out or sort by arrival of the requests.
- Order by the client ID.
- Order by transaction priority.
- Order the list by the number of locks held by each transaction. There are two possibilities:
  - Transactions with fewer number of locks go first.
  - Transactions with greater number of locks go first.
- Serve the read requests first.
- Split up the read requests according to the multiprogramming capabilities.
- Order requests such that the total distance traversed by the messages is minimized.

The first two ordering criteria are simple, and it is expected that they will perform the worst. Using one or more of the remaining criteria, a cost function may be developed, which may be minimized to obtain the best performance. We intend to evaluate such cost functions to determine the circumstances under which the best performance may be obtained. The cost function minimization will certainly require more computations and hence more processing. However, it should be noted that the processing is done while the server is *waiting* for the data to be returned, thus making efficient use of the CPU cycles.

The data structure for the forward list for each data item will be a list with appropriate markers to delimit the parallel shared accesses and the serial exclusive

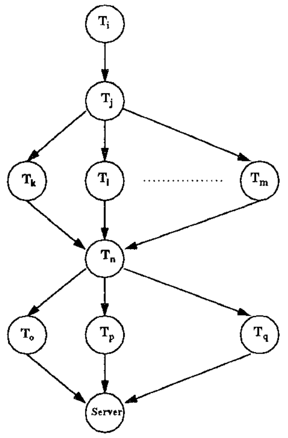

Figure 5: Structure of the Transaction Precedence Graph for a Data Item

accesses. Each list entry will contain the pair : the transaction ID and the corresponding client site ID. For the example in Figure 3, the forward list sent by the server to clients 1 and 2 is: [(T_1,1),(T_2,2)], (T_3,3), where the entries between [..] are the shared accesses.

Second Step: In the second step, the initial forward list (created in the first step) is re-ordered with deadlock prevention in mind. It is well known that the 2PL protocol suffers from deadlocks. Two or more transactions are said to be in a deadlock when neither of the transactions can proceed because at least one of the locks required by each of the transactions is held by one of the other transactions.

Deadlocks can be prevented if in each of the forward lists, the order of the transactions is the same. Formally, the forward list for each data item can be represented by a transaction precedence graph, which need to be made consistent. That is two transactions T_i and T_j must follow the same order $< T_i, T_j >$ or $< T_j, T_i >$ in every precedence graph involving T_i and T_j. 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 is termed a predecessor transaction, and a transaction that immediately follows is termed a successor transaction. Note that the precedence graph is consistent with the lock granting order and hence consistent with the serialization order. The transaction precedence graph obtained after optimizing the cost function will have a general structure as given in Figure 5. Each transaction in control of the data item may pass the data item to one or multiple transactions. At any time, there may be one or more concurrently executing transactions. The stages with just one transaction refer to exclusive access by the transaction while the stages with multiple transactions refer to shared access by multiple transactions in parallel. It should be clear from Figure 5 that a transaction may have multiple successors and predecessors and the set of successors/predecessors must be determined from the

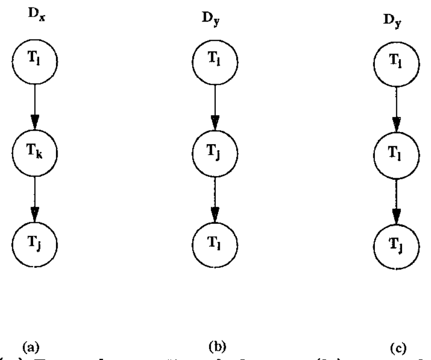

Figure 6: (a) Precedence Graph for D_x (b) Precedence Graph for D_y (c) Revised Precedence Graph for D_y

precedence graphs of all data items accessed by the transaction. The last node in the precedence graph is the server, so the last client(s) can return the data item to the server, which then serves the next window.

Also, to make efficient use of all the resources, sometimes the transaction precedence graph for a data item may need to be further re-ordered while maintaining its consistency achieved in step 2. The idea here is to minimize the time a site is waiting for a data item or to allow multiple data to be combined in a single large message taking advantage of the huge bandwidth of the network. For an example, consider Figures 6(a) and (b) which depict parts of the precedence graphs for two data items D_x and D_y. Since transaction T_j has to wait for D_x until T_k releases it, so it might make more sense to re-order transactions T_j and T_l in the precedence graph for D_y as well. The revised precedence graph is depicted in Figure 6(c).

Again, it should be stressed that all of these computations and reordering are done while the server is waiting for the data items to be returned from the clients in the previous window. Thus, these computations do not add to the transaction response time, and in fact increase the utilization of the server CPU.

3.3 Related Work

In the concurrency control mechanism described above, the DB server acts as a dispatcher for the data items. This can also be viewed as a scheme where the primary copy of each data item is stored at the DB server, and a floating copy of the data item migrates from client to client under the supervision of the server. This scheme has a similar flavor to the send-on-demand scheme proposed in [6]. However, in that scheme, the data items were migrated from site to site according to the demand generated. With the location change, the ownership of the data item is transferred to the new site, thus making the recovery mechanism difficult (owing to the distribution of the log records). The scheme proposed in this paper differs in that each data item is owned by a specific DB server. In the Mariposa database system [13], one of the performance-enhancing criteria proposed was that the ownership of data items should not be fixed, and it was acknowledged that in doing so, the recovery operation would become very difficult. However, the motivation there was different from the one here, that of reducing the effects of communication propagation latency. In the next section, a possible recovery scheme is proposed.

4 Failure Recovery in GNDB

Thus far, we have not considered failures. The proposed scheme can be made resilient, if we assume that the server as well as each client support stable storage. 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. Without 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, the DB 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.

A potentially more efficient 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 need 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 concurrency control and recovery scheme might be possible at the cost of more messages with the advantage of less rounds. In the sequel, we consider such a recovery scheme that attempts to reduce the overall cost of recovery by distinguishing between reliable and unreliable sites.

4.1 An Adaptive Recovery Scheme

Recovery is very difficult in a situation where data items may migrate from site to site [13, 14]. 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 only site failures need to be considered.

Every site may be dynamically classified into two broad types: *reliable* and *unreliable*. The only difference between a reliable and an unreliable site is that if a reliable site 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 site 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 site executing the transaction. Below we first discuss the recovery operation for reliable sites, and then follow up with the more interesting case of unreliable sites. The server records each site as a reliable or unreliable site. If there is no information available on a particular site, the server may adopt a pessimistic approach and assume that the site is unreliable.

A. Reliable Sites: Since reliable sites are expected to be able to recover from failures relatively quickly, reliable sites are assumed to support stable storage, combined with an efficient write-ahead-logging (WAL) scheme, e.g., Aries [15]. 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 acknowledgement message to its predecessor. A transaction can be committed as soon as the part of the log pertaining to the transaction is on the stable storage. Using this scheme, if the site fails, until it comes back up again, the data items at that site will be unavailable, but not lost. The important thing to note is that no communication (and hence propagation latency costs) is required with other client sites to commit a transaction. Of course, the site will have to synchronize its cache with the server, but can do so after committing the transaction, thereby not adding to the transaction response time.

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 sites that require the data items immediately after the current transaction on the failed unreliable site. These sites are referred to as the *successor* sites of the transaction and the transaction as the predecessor of these sites. Note that each successor site of a transaction may require only a subset of the data items currently held by the transaction. 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-images of committed data items, or the before-images of uncommitted 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. In the following, we propose an *atomic commit protocol* [11] that allows the set of successors to reach a consistent decision, and gain access to the correct data.

When a predecessor transaction of an unreliable site commits, it sends the data item to its successor site as well as to the successor of its successor for that data item. Thus, all successors of an unreliable site obtain the before-images of the data items required by them, as well as learn the identity of their unreliable predecessors. 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 to the server as well as to the set of its successors[3]. 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 or the server, 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

[3]If a site is concurrently processing n transactions, it will be part of n successor site sets. Here it is assumed that $n = 1$. The successor set is constructed from the forward lists.

the successor sites or the server 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, 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.

If an unreliable site fails before or after sending the after-images, the successor sites that do not receive the after-images within a specific time-out may enquire as to the status of the transaction from the server[4]. If the server has received the after-images (and has acknowledged the message), it sends the after-images to all the successors. Otherwise, the server initiates a voting to abort the transaction at the failed site.

In the first step of the voting, the server sends an enquiry message to all the successors of the failed site, along with the successor set. Even if one successor site has received the after-images, it will send this information to all the successors and the server, allowing the successors to 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 (that was sent by the server or a previous client site) will be used.

When the failed site recovers, and sends the after-images to successors and the server, the receiving sites respond with an abort message which causes the transaction at the failed site to abort. Thus, even under the very improbable circumstance that *all* the messages containing the after-images are lost[5], the successors to the current transaction will be able to proceed on the assumption that the predecessor transaction was aborted. The recovery algorithm including the voting protocol for unreliable sites is specified in Figure 7.

Example: Consider the same example as discussed before with three transactions, all accessing the same data item D_x in exclusive mode. Further, site 2 requires data item D_y, which is then required by transaction T_4 at client site 4. Assume that the first client site involved is reliable, and the other three sites are unreliable. Site 1 thus uses the simpler recovery scheme for transaction T_1 with local WAL commit procedure. If the client site 1 fails during the recovery process, all other successor sites (in this case, site 2) will be blocked. However, since site 1 is a reliable site, it will recover from the failure shortly. The more involved recovery case is when dealing with site 2.

Since site 2 is unreliable, client 1 will send its committed data D_x to both sites 2 and 3 as well as the

[4]Note that the successor that does not receive the after-images, does not know the identity of the other successors.

[5]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.

- If the successor is unreliable, send committed data to
  - the server, and
  - the next two successors on each forward list.
- 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 is declared to be failed,
  - Send enquiry message (ENQ) to the server with the failed predecessor site ID (pred-ID).
- On receiving an ENQ message,

⋆ if the server has received after-images of a data item from pred-ID (and has acknowledged the data),
  - it will broadcast the released data to all the successors of pred-ID.

⋆ if the server has not heard from pred-ID, then the server initiates a vote-to-abort the transaction at pred-ID.

Voting scheme

Phase 1: Server 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 predecessors and successors of pred-ID.
- Otherwise, it sends its yes vote-to-abort to the server and the predecessors and the successors of pred-ID.

Phase 2: All successors and predecessors of pred-ID decide to abort the message after getting *all* yes-votes from the successors of 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 acknowledgements from atleast one of the successors or the server. If it receives an abort message in response, it aborts its transaction.

Figure 7: Unreliable Site Recovery Protocol

server. When ready to commit transaction T_2, site 2 will send the after-images of data items D_x and D_y to *both* its successors (site 3 and 4) and the server. Assuming that site 3 is physically closer to site 2 as well as lightly loaded, the acknowledgment from site 3 reaches site 2 first. At this point, T_2 commits. If for some reason, site 4 does not receive the after-images, within a timeout period, it can query site 2, 3, and the server to obtain the after-images.

If site 2 fails after sending the after-images, and both successors and the server do not receive the messages containing the after-images, the server will initiate the voting procedure with the successors and predecessors of the failed site 2. Since no site has the after-images, they will all decide to abort the transaction at site 2, and use the before-images to process their respective transactions, thus bypassing T_2. In the meantime, if site 2 recovers, and retransmits the after-images, both successor sites and the server will return abort messages. T_2 will then be aborted.

5 Conclusions

In this paper, two novel data sharing and access protocols, known as APPLODS (Algorithms for Propagation Latency Optimization in Distributed Database Systems) that are suited to the gigabit WAN environment have been proposed. The purpose of the paper was to motivate the need for new schemes in a gigabit environment, and propose a possible latency reduction mechanism. Previous research has produced possible data sharing schemes that would scale to the gigabit environments, but the recovery process in those cases is extremely difficult. One of the major contributions of this paper is the development of a practical recovery process in conjunction with the new data sharing scheme. For brevity, the above discussion made a number of simplifications such as a single transaction per site, and a single type of lock, and omitted to discuss a number of extensions such as the possibility of the server to dynamically extend a forward list. We believe these as well as other optimizations will further enhance the performance of the proposed scheme taking advantage of the new characteristics of a high speed network.

An important parameter in this scheme is the window size, which will directly have performance implications. Having a small window size will cause a larger number of sequential message transfers, thus reducing the performance level. Also, the probability of deadlock increases with a decrease in window size, since the opportunity of re-ordering the granting of locks within a window to minimize deadlocks is also reduced. With a large window size, the first transaction is delayed by a maximum of the request collection duration, which delays the other transactions as well. Thus, an optimal value of the window size needs to be used. Also, depending on the transaction rates, and its QoS requirements, the window size may have to be dynamically adjusted. Future work planned includes simulation studies to demonstrate the expected performance improvement, and also extend the proposed schemes to other configurations of client-servers.

References

[1] R. Ramaswami, "Multiwavelength Lightwave Networks for Computer Communication," *IEEE Communications*, vol. 31, pp. 78–88, Feb. 1993.

[2] L. Kleinrock, "The Latency/Bandwidth Tradeoff in Gigabit Networks," *IEEE Communications*, vol. 30, pp. 36–40, April 1992.

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

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

[5] J. D. Touch and D. J. Farber, "The Effect of Latency on Protocols," *Proceedings of the ACM SIGCOMM*, 1994.

[6] S. Banerjee, V.O.K. Li, and C. Wang, "Distributed Database Systems in High-Speed Wide-Area Networks," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 617–630, 1993.

[7] S. Banerjee and V.O.K. Li, "Application Protocols for Wide Area Gigabit Networks," *Proceedings of the Gigabit Networking Workshop*, 1994.

[8] M. Carey, M. Franklin, M. Livny, and E. Shekita., "Data Caching Tradeoffs in Client-Server DBMS Architectures," *Proceedings of the ACM SIGMOD*, pp. 357–366, 1991.

[9] Y. Wang and L. Rowe., "Cache Consistency and Concurrency Control in a Client/server DBMS Architecture," *Proceedings of the ACM SIGMOD*, pp. 367–376, 1991.

[10] K. P. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The Notion of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, vol. 19, pp. 624–633, November 1976.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

[12] W. Stallings, *Data and Computer Communications.* Macmilan, 1991.

[13] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson, "Mariposa: A New Architecture for Distributed Data," *Proceedings of International Conference on Data Engineering*, pp. 54–65, 1994.

[14] S. Banerjee, *Distributed Database Systems in High Speed Networks.* PhD thesis, University of Southern California, August 1993.

[15] 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*, vol. 17, pp. 94–162, March 1992.