Simplifying Distributed Database Systems Design by Using a Broadcast Network

Jo-Mei Chang

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

Atomic broadcast and failure detection are powerful primitives for distributed database systems. In the distributed database system LAMBDA, they are provided as network primitives. In this paper, we show how atomic broadcast and failure detection simplify transaction commitment, concurrency control, and crash recovery. Specifically, we give a simple *two-phase non-blocking* commit protocol; whereas three phases are required in a point-to-point network. We also give a simplified read-one/write-all update algorithm for replicated data and an easily implemented log-based recovery algorithm providing uninterrupted transaction processing.

The benefits of performing the atomic broadcast and failure detection at the network level are also discussed. Performing these functions at the network level not only simplifies database protocols but also better utilizes the broadcast network: fewer messages are transmitted. Comparisons between LAMBDA and existing distributed database systems are also made.

1. Introduction

A project called LAMBDA at AT&T Bell Laboratories, Murray Hill, is investigating the impact of a broadcast network on the design of distributed systems. Specifically, LAMBDA aims at exploiting the features of the broadcast network to simplify the design of distributed database systems. In this paper, we present some of the results obtained in LAMBDA.

In a broadcast network, either a local area network with broadcast feature or a satellite network, messages can be sent to a group of sites by using one broadcast message. In contrast to a point-to-point network where messages must be sent individually to each site, the broadcast network provides an economic way of communicating among a group of sites. Furthermore, for certain types of broadcast networks, such as Ethernet [METC 76], there exists a unique sequence among the messages transmitted in the network: when messages are not lost, if broadcast message m_1 is received at one site before a message m_2, m_1 is received at all sites before m_2. In contrast to a point-to-point network, where messages from different sources can arrive at a site in reverse order in which they were sent, this type of broadcast network provides a more synchronized communication environment.

A common task in a distributed database system is to send the same information to many or all the sites participating in a given database transaction, for example, transaction update information. Reaching agreement is another frequent task in a distributed system. Transaction commitment, where sites must unanimously decide to commit or abort a transaction, is the best known example of this. Agreement also lies at the heart of many update protocols for replicated copies: here each process must agree on whether a given copy is operational or not. In all the above, an efficient broadcast facility simplies the task while reducing its cost.

One major source of complexity in designing distributed database systems is due to communication failures and site failures. Consequently, database protocols, such as fault-tolerant commit protocols, are carefully designed to guard against these failures. Fault-tolerant protocols tend to be much more complex than their non-tolerant counterparts; in addition, fault-tolerant protocols, performing similar functions, must be introduced in several modules in a typical database system. It is not surprising then that fault-tolerance is a major source of complexity in a distributed database system and is a significant factor in system performance.

The broadcast network, on the other hand, cannot be fully utilized by the application programs due to the unreliable communication medium: broadcast messages may not arrive at all intended recipients or may be lost altogether. The cost and complexity of fault-tolerance suggests that the first step toward both exploiting broadcast networks and simplifying distributed database design is to provide a *reliable broadcast communication environment that detects failures.* Once the underlying communication network is

made reliable and made to detect site failures, the responsibility of the distributed database can be reduced and the system simplified. This is the approach taken in the LAMBDA database system [CHANG 83b].

In this paper we first describe the features of the reliable broadcast environment provided in LAMBDA, and then discuss how this environment can simplify the design of a distributed database system, focusing on issues of crash recovery and concurrency control. Examples are drawn from a number of systems, supporting nonreplicated and replicated data, with emphasis on the LAMBDA system. Specifically, in LAMBDA, non-blocking transaction commit can be provided by a simple two-phase commit protocol. In contrast, two-phase commit protocols are known to lead to blocking situation in the presence of multiple failures in a point-to-point network. Treating replicated data is another important but complex issue in designing distributed database systems. In LAMBDA, the read-one/write-all update strategy and the log-based recovery mechanism can be easily implemented and thus provides high data availability, automatic recovery from site failure, and uninterrupted transaction processing during recovery.

The remainder of the paper is structured as follows. In Section 2, we describe a reliable broadcast network environment – LAMBDA NET – and the network interface to database processes. Only the properties of the reliable broadcast network environment are given. The descriptions of the LAMBDA NET protocols – a reliable broadcast protocol and a reformation protocol – are given in the Appendix A and B. Section 3 is the heart of the paper, we discuss database transaction commit, concurrency control and crash recovery issues. Section 4 compares LAMBDA with other existing distributed database systems. Section 5 gives our conclusion.

2. A Reliable Broadcast Network

A number of local area networks provide a broadcast feature [METC 76]: broadcast/multicast messages can be received by all sites in the broadcast group. However, some intended receivers may lose broadcast messages because of transmission errors or buffer overflows. For this type of local network, a reliable broadcast environment can be established by using a reliable broadcast protocol between the application program and the broadcast network. This section describes the features of the reliable broadcast network environment [CHANG 83a, 83c], called LAMBDA NET, currently being implemented on an Ethernet for the LAMBDA system.

2.1 Services Provided by LAMBDA NET

A system is *k-resilient* if it can continue normal operations in the presence of k or fewer failures. Under the assumption that site failures are benign – that is, sites do not lie about their state – LAMBDA NET employs a strong notion of k-resiliency: it always continues message processing when k or fewer site failures occur and can detect when it is safe to continue processing when more than k failures occur. The system often continues processing in the presence of many more than k failures.

LAMBDA NET provides the following features:

Atomic broadcast: Ensuring two very strong properties:

(1) failure atomicity (all-or-nothing property) – if a message is received by an application program at one site, it will be received by application programs at all operational sites; and

(2) message synchronization – if message m_i is received at one site before m_j, it is received at all operational sites before m_j.

System wide logical clock: Each broadcast message is timestamped with a unique incrementally increasing timestamp. A message with a larger timestamp can not influence the issuing of a message with a smaller timestamp.

Failure detection: When failures or recoveries are detected, a site status message is broadcast. (This message looks like any other broadcast message, although it is implemented differently.) Whenever a site is declared not operational, it will not (successfully) broadcast any messages until a site status message announcing its recovery is sent. Therefore, LAMBDA NET never delivers an obsolete message from a failed site.

Network partitioning announcement: Whenever more than a majority of sites have failed, a network partitioning is announced. This ensures that application programs are aware of the possibility of network partitioning and take precautionary measures, e.g., to block processing completely or to be willing to back out when the communication link is repaired [DAVID 81].

When a broadcast message is first received by a site, it is considered *uncommitted* and can not be read by an application process. It remains uncommitted until the network can guarantee the two atomic broadcast properties given above. The message is then placed in a message queue for the intended process and is thereby *committed.* The above features apply only to committed messages. Property (1) under atomic broadcast can now be more rigorously stated: if any site commits the message, then all operational sites commit the message. Message queues for committed messages are not stored in stable storage; therefore, a process will receive a committed message only if its host site remains operational long enough for it to read the message from its queue.

The guaranteed delivery feature of atomic broadcast is stronger than most reliable broadcast protocols (for example [SCHNE 82]), including some types of Byzantine Agreement protocols [LAMPO 82]. The latter protocols provide only *interactive consistency,* sites operational at the end of the protocol either all commit or none commit the broadcast message. Interactive consistency allows a site that fails during the execution of a protocol to reach a different conclusion than those that remain operational

until the end. This is not acceptable for many applications. The case where the failed site commits the message while the operational sites do not is particularly dangerous, especially when the message can precipitate irreversible actions such as the dispensing of money in an automated teller. Atomic broadcast does not allow this.

Message sequencing is another important aspect of atomic broadcasts, simplifying the design of several database protocols. This is especially true when failures and recoveries are synchronized with normal messages.

The above features are defined with respect to a broadcast group, which is a static set of sites. LAMBDA NET allows any number of broadcast groups; each one sequences its messages, maintains its own clock, and detects failures. Activities from different groups are not coordinated. A site may belong to any number of broadcast groups.

2.2 Implementation of LAMBDA NET

LAMBDA NET is currently built on top of an Ethernet using the "reliable broadcast protocol" and the "reformation protocol." The reliable broadcast protocol maintains the logical clock and provides the atomic broadcast facility. Whenever a failure is detected, the broadcast protocol is interrupted and control passes to the reformation protocol, which sits at a lower (logical) level in the system. The reformation protocol determines the new network configuration, including the status of all sites. When it finishes, the broadcast protocol resumes.

Basically, the reliable broadcast protocol uses a combination of positive acknowledgements, negative acknowledgements and token passing scheme. Only the token site is required to acknowledge the receipt of a broadcast message: each acknowledgement carries an incrementally increasing sequence number, called a timestamp. Based on the sequence number, a non-token site requests missing acknowledgements and thus missing broadcast messages from the token site. The token site responsibility is rotated among all operational sites in the broadcast group. Failure in the system is detected using low level timeout. Since the token is passed among operational sites, any site failure is guaranteed to be detected within N token transfers, where N is the number of operational sites in the broadcast group. Brief descriptions of the reliable broadcast protocol and the reformation protocol are given in Appendices A and B. Detailed descriptions and analysis of these two protocols can be found in [CHANG 83a, 83b] and [MAXEM 84].

The overhead in implementing LAMBDA NET is surprisingly low: it requires only one acknowledgement per broadcast message (instead of one acknowledgement per site per broadcast message). It also requires only limited storage for retransmission messages: for a broadcast group with N sites, the protocol can operate with only N-1 messages and acknowledgements retained. Note that the above overhead is independent of k, the resiliency parameter. Extra resiliency in LAMBDA NET is obtained by introducing message delays rather than transmitting extra messages, as described in Appendix A. For more complete analysis, please refer to [MAXEM 84].

Although LAMBDA NET is currently implemented on top of Ethernet, it can be efficiently implemented on top of other types of broadcast networks, such as a ring network: the LAMBDA NET features described herein are not specific to the Ethernet.

2.3 A Database Interface

In the LAMBDA distributed database system, all sites participating in database operations are treated as one large broadcast group. Consequently, the entire database system sees a total ordering over all broadcast messages and has access to an unique global logical clock. A message is usually addressed to only a subset of these sites, a broadcast subgroup. Since there is only one relevant broadcast group, we will let the term "group" refer to the broadcast subgroup.

Transparent to database processes and lying between them and LAMBDA NET is a *message filter*. At each site, the filter process: (1) removes messages not addressed to a process at the site, (2) dispatches messages (in sequence) to the intended database processes, (3) forwards site status messages (in sequence) to all database processes.[1] Because processes within a site may operate asynchronously, each process maintains its own status information. (The message filter process also maintains a site status table for its own use). Note that each process (including the filter process) receives all messages, including site status messages, in increasing timestamp order.

In the following discussion, a simplified failure assumption is used: process failures result only from site failures. This assumes that processes do not die individually because of errors or user actions. If processes within a site can fail independently, mechanisms can be added to monitor process status, e.g., the message filter process can be devised to monitor critical database processes and to announce detected failures.

The network interface to database processes consists of a number of broadcast primitives and, of course, routines for standard point-to-point communication, the latter not providing message synchronization. The principal broadcast primitive is the unadorned atomic broadcast, denoted *brdcst*$(MSG, Group_i)$, where $Group_i$ is the broadcast subgroup that should receive message *MSG*. *Brdcst* is a function returning the subset of $Group_i$ that is defined by LAMBDA NET to be operational when MSG is sent. Let us call this subset the *recipients* of *MSG* . This subset may contain sites that failed prior to the sending of MSG but whose failures have not been detected by the network. (This facility is provided by the message filter process at the transmitting site. The filter process defines

1. In a sophisticated design of the filter process, only relevant recovery or failure information is sent to each process.

the set of operational sites according to the site status table at the time MSG is committed. Because messages, including the site status messages, are committed in the same sequence everywhere, this information is consistent at all sites).

Even though the set returned by *brdcst* is a superset of the sites actually committing the message, this information is still useful. For example, the broadcaster can safely wait for responses from all recipients because each one either will remain operational long enough for the receiving process to read the message and respond or will fail (or has already failed but not yet been detected) and the network will deliver a failure notice (in the form of a status message) to the broadcaster. Note that, although the exact set of operational sites in the broadcast group to commit a message could be determined, at some expense, when the message is acknowledged; this information is not very meaningful since a committed yet unprocessed message is lost when the site fails.

Broadly speaking, there exist two types of acknowledgements:

1. a low level acknowledgement sent by the network to acknowledge that a site has received a message;
2. a high level acknowledgement sent by the process to acknowledge that a site has processed a message.

The atomic broadcast primitive is provided using low level acknowledgements. In many cases, such as announcing a unilateral decision, this primitive suffices. In other cases, such as to coordinate a task among a set of remote sites, high level acknowledgements are needed, since each site must report its current status. High level acknowledgements are also needed if a broadcaster is to know which processes has actually processed a committed message. To coordinate among a set of sites and solicit high level acknowledgements (responses), LAMBDA provides two higher level broadcast primitives: a *broadcast-all protocol* and a *broadcast-select protocol.*

In the broadcast-all protocol, a coordinator broadcasts one request to a broadcast group and expects responses from each of the recipients of the request. A response here will either be a reply sent by the recipient or a failure notification sent by LAMBDA NET. Broadcast-all returns to the sender the full set of responses. In a distributed database system, the broadcast-all protocol may be used in phase I of a two-phase commit protocol.

In a broadcast-select protocol, the coordinator broadcasts one request and solicits only M non-null responses. Often M is one. The protocol returns the set of M responses or returns *ERROR* if it becomes clear that the request cannot be satisfied. A null response indicates that the recipient can not respond to the request in a meaningful way. Null responses are often used by processes recovering from a recent failure and is necessary to prevent delaying the coordinator unnecessarily. This protocol is useful in performing resource allocation. In a database system, a broadcast-select protocol may be used when a coordinator wants to read from replicated data copies.

The syntax for these primitive is *Brdcst_All* (MSG,$Group_i$) and *Brdcst_Select* (MSG,$Group_i$,M).

3. Distributed Database Design Issues

Atomic broadcast and failure detection are major primitives provided by LAMBDA NET. These are convenient and powerful primitives for distributed systems, and we now demonstrate this by showing how the LAMBDA NET environment simplifies the problems of transaction commit, concurrency control, and crash recovery. Many of the examples are drawn from the LAMBDA distributed database system [CHANG 83b], which is being implemented on top of LAMBDA NET and which supports replicated data.

For transaction processing, we assume the TM-DM model of [BERNS 81]. A transaction manager (TM) and a database manager (DM) reside at each site. The transaction managers supervise user queries and translates them into commands for database managers. Each transaction is executed by a single TM. A database manager maintains that part of the database stored at its site. A DM may be concurrently processing transactions from several TMs.

The TM-DM model is used solely for expository convenience, none of the following results are specific to this model.

3.1 Transaction Commit

At the end of processing a transaction, a commit protocol is invoked to either commit or abort the transaction at all participating DMs. The most used protocol is the two-phase commit. Briefly, it is [GRAY 78]:

Phase 1: The coordinating TM sends the "prewrite" message to all DMs updating their databases. Each DM prepares to write into the permanent database its changes, acquiring any needed locks and logging recovery data. Each DM responds with "accept" or "reject."

Phase 2: If all DMs accept, then the TM sends "commit" to all DMs; otherwise, it sends "abort."

The major weakness of this protocol is that it is not resilient to coordinator failures: if the coordinator and all DMs that know the decision (i.e., commit or abort) fail, then the protocol is blocked and the transaction can not be completed until some processes recover.

Four-phase [HAMME 80] and three-phase [SKEEN 81] nonblocking commit protocols are known, but these are more expensive, by 50 to 100 percent, in time and messages, and much harder to implement. They do have the feature that the transaction can always be correctly completed, without waiting on failed processes, if k or fewer failures occur (where k is the resiliency parameter).

Using the LAMBDA NET environment, the simple two-phase commit protocol of Figure 1 provides the same

COORDINATING TM:

Phase I: Msgs := Brdcst_All("Prewrite" $Group_i$);

Phase II: if all Msgs = "accept"
then Brdcst("Commit" $Group_i$);
else Brdcst("Abort" $Group_i$);

PARTICIPATING DMs:

Phase I: wait for either a prewrite from TM or a failure notice;
if "prewrite" is received
then prepare to commit;
respond to TM with either "accept" or "reject";
else abort the transaction;

Phase II: wait for either a Commit, Abort, or TM failure notice;
if "commit" received
then commit the transaction;
else abort the transaction;

Figure 1. A nonblocking 2-phase commit protocol using LAMBDA NET.

nonblocking property. Let's examine why this protocol works, concentrating first on the second phase. The atomic broadcast feature ensures that if any DM receives a commit message (before the failure notice), all operational DMs must receive the message (before the failure notice). Clearly the strong all-or-nothing property of atomic broadcast is necessary here. In particular, interactive consistency – the requirement that only operational DMs must agree – is too weak. To see why, consider the case where the TM fails in phase II while sending "commit." Broadcast protocols providing only interactive consistency would allow one DM to receive the commit message, commit the transaction, and then fail; meanwhile, the other DMs could agree that no message was sent. This results in an inconsistent database.

Phase II requires not only the all-or-nothing property of atomic broadcast but also the message synchronization property, especially, the synchronization of failure notices with other messages that LAMBDA NET provides. Consider what may happen if failure notice synchronization was not provided. Suppose that the TM fails immediately after broadcasting "commit" in phase II and failure notices are sent to all operational DMs. Suppose further that some but not all DMs receive the failure notice before the commit message. Either these DMs can abort resulting in an inconsistent database or they can wait for an appropriate amount of time to see if a message from the TM is forthcoming. The problem with the latter approach is determining "an appropriate amount" of time. This requires bounding message delivery time, which is hard to do; furthermore, upper bounds are assured with high probability only, not with certainty.

Phase I of the protocol illustrates the use of the broadcast-all protocol. This is certainly a convenient use of the protocol, but the features provided by the protocol are stronger than necessary: it is not critical that all DMs receive the message if the TM fails in this phase, since in this case there is only one possible outcome – to abort – and all DMs will eventually conclude this. An interesting question is whether there is a cheaper alternative. Suppose the TM uses unreliable broadcast. Since there is no guarantee that all sites receive the message, it will now need to use timeouts or low level acknowledgements; otherwise, it could wait indefinitely for a reply from a process that never received the message. Timeouts at the process level are hard to tune. Using low level acknowledgements requires more messages, yet is not necessarily faster than the implementation of atomic broadcast described in Appendices A and B.

It is not surprising that atomic broadcast greatly simplifies nonblocking commit protocols: the last two phases of the three-phase commit protocol is essentially implementing atomic broadcast at the database process level [SKEEN 83b]. However, in the case of LAMBDA NET, the atomic broadcast is implemented at the network level, where it is cheaper and its complexity can be hidden from application programmers.

3.2 Replicated Databases

Consider now a database system supporting replicated data. For each logical data item X, there exists a number of physical copies $X_1, X_2, \cdots, X_m$, where $m \geq 1$. For our purposes, the unit of replication is not important. Replication complicates concurrency control and adds a new problem, that of maintaining consistency among copies.

3.2.1 Consistency among Copies. Transaction managers and other software components at a comparable level of the system normally deal with logical data items. Requests to logical item X must be mapped into requests to appropriate copies of X. Similarly, responses from copies of X must be mapped back into a single response. These mappings must preserve the intuitive notion of correctness: the multiple copies of X should behave as if they were a single copy [BERNS 83].

A popular scheme for maintaining one-copy behavior, which has been proposed for ADAPLEX [GOODM 83], distributed-INGRES [STONE 79], and LAMBDA [CHANG 83b], is what we will call the *active group* method. To read X, a transaction manager reads from *any operational* copy of X. To write X, a transaction manager writes *all operational* copies of X. By operational copies, we mean copies that include the most recent updates, in contrast to failed or recovering copies. This method yields high read and write availability: as long as one copy is operational, the logical item is available for both reading and writing.

Implementing the active group method requires transaction managers to have consistent copy status information. Consider what may happen if two transactions T_i and T_j have inconsistent views of operational copies. Suppose T_i considers X as having two operational copies X_1 and X_2, and T_j considers X as having only one operational copy X_2.

T_i and T_j can simultaneously read and write X: T_i reads from X_1 and T_j writes to X_2. Because the copy status information is itself dynamically changing and replicated, the active group method is difficult to implement in a general network environment.

Consider now implementing the active group approach in the LAMBDA NET environment, assuming that the copies do not migrate and location information is generally available. The problem of maintaining a consistent view of operational copy status among transaction managers is partially solved by using site status information. Since all processes receive the site status messages in the same sequence, all processes agree on the set of operational sites that have a copy of X. Processes can therefore use the broadcast primitives of LAMBDA NET to reach all operational copies. A broadcast will also reach non-operational copies, such as recovering copies, that reside at operational sites; but these copies can ignore the request or, if a response is required, return a null response.

Figure 2 describes in detail the read-one/write-all protocols implemented in the LAMBDA database system. A two phase locking protocol [GRAY 78] is used to ensure correct execution of concurrent transactions. Depending on the lock status, a read or write request on operational data is answered either "yes" or "no". A recovering copy always returns a null response. (Due to the underlying failure detection facility, a slow response, due to a transaction waiting for a lock, can be differentiated from a failed site). Distributed deadlock detection is required but this is an unrelated issue.

Let $Group_X$ represent the set of sites that have a copy of X. For a read operation on X, the transaction manager can arbitrarily select an operational site, send its request, and hope that the site has an operational copy. Alternatively, a transaction manager can use a broadcast-select protocol to broadcast the request to $Group_X$ and wait for the first response from an operational copy of X. For a write operation on X, a transaction manager uses a broadcast-all protocol to send the request to $Group_X$. It will receive the set of responses from all operational sites in $Group_X$, from which it can determine the operational copies (they return non-null responses) and verify that all were able to perform the request. (Note that although the set of operational sites in $Group_x$ may vary from time to time, this problem has been taken care of by the broadcast-all protocol: all sites that are operational at the time the request is sent, as returned by the *brdcst* function, will respond with yes, no, null, or failed (sent by the network)).

3.2.2 Concurrency Control. A problem arises when the active group approach is used with two-phase locking. Normally locks are held only at the copies that were actually accessed. Hence, if transaction T reads X, it locks only the one copy actually read. This single read lock prevents any other transaction from writing X since to write all operational copies must be locked. If however the copy locked by T fails, then this is no longer true. Lacking locks at any operational copy of X, T has effectively lost its lock on the logical data item X.

Read One protocol:

```
send("read X") to an arbitrary operational site in GroupX;
if (response = "null") {
        Res := Brdcst_Select("read X", GroupX, 1);
                if (Res = "yes")
                        read is successful;
                else read is unsuccessful;
        }
else if (response = "yes")
        read is successful;
else if (response = "no")
        read is unsuccessful;
```

Write All protocol:

```
Msgs := Brdcst_All(write X request, GroupX);
if ("no"∈Msgs)
        write is unsuccessful;
else if ("yes"∈Msgs)
        write is successful;
else    /* all Msgs = "null" or "failed" */
        write is unsuccessful;
```

Figure 2. Protocols for read and write operations.

Whenever a transaction loses a lock on a logical item before it completes, that transaction has, in effect, violated the two-phase locking constraint. It is easy to construct examples where losing locks in this way leads to nonserializable executions [BERNS 83]. Note that problems can only arise with lost read locks, not write locks: losing all write locks for an item means that all copies are non-operational and, hence, the item is inaccessible.

A correct two-phase locking strategy to go with active groups is to check all read locks at commit time. If a transaction owns a read lock from a site that has failed since the read lock is granted, the transaction must be aborted. This check is easily performed in LAMBDA NET using the site status information provided. Note that the correctness of this scheme relies on failure notices arriving in the same order at all sites. [BERNS 83] proves a similar scheme correct.

Although failures pose no problems regarding write locks, recoveries do. If transaction T writes X, it locks all operational copies. This prevents any other transaction from reading (or writing) X since to read (or write) one (or all) operational copy must be locked. If however a copy of X changes its status from recovering to operational, then this is no longer true. This copy is now available for reading and can grant a read lock on the logical item X to a different transaction. Lacking a lock at an operational copy, T has not effectively write-locked the logical data item X. T has to be aborted, unless the recovering copy happens to carry the correct lock. A correct recovery strategy for the active group approach is to require a recovering copy to *inherit* write locks from other operational copies of X before it becomes operational.

So far, we have discussed only one type of concurrency control mechanism: two-phase locking. The LAMBDA NET environment can be used to simplify other types of concurrency control mechanisms as well, e.g., timestamp concurrency control [BERNS 80]. The timestamps provided by the LAMBDA NET can be used directly as timestamps for transactions. As pointed out in [CHAN 83b], if the SDD-1 concurrency control algorithm is used in the LAMBDA NET environment, a number of deficiencies such as the requirement for "send null" and "null write" messages can be eliminated.

3.3 Crash Recovery

When data is replicated, upon recovery, the simplest method is to copy the entire data from an operational copy. However, this method is very expensive. Popular and much cheaper schemes use *logs* for recovery. For each physical copy X_1, a log file, $\log(X_1)$, records the sequence of updates (prewrite and commit messages, as described in the commit protocol) that have been applied to X_1. To recover, a recovering copy must identify the set of missing updates in the log of an operational copy of X and apply these updates.

Generally, log-based recovery is very messy because log files at different sites may record updates in different orders. This can happen because each DM may be concurrently processing several transactions and updates from different transactions may be executed in a different sequence at each site. The consequence of this is that it is difficult for a site to concisely identify the updates it is missing. To illustrate the problem, consider the following sequence of updates recorded in two log files $\log(X_1)$ and $\log(X_2)$, ignoring momentarily the | mark.

$\log(X_1)$ = { ..., Prewrite_j, Commit_j, Commit_i, ...}
$\log(X_2)$ = { ..., Commit_i, | Prewrite_j, Commit_j, ...}

In these log files, updates from transactions T_i and T_j are recorded in different orders. Now consider what happens if X_2 fails at the | mark. The last log record appearing in $\log(X_2)$ is Commit_i. Upon recovery, how does X_2 request the updates that it missed? If X_2 requests only what is in $\log(X_1)$ after Commit_i, the last update that X_2 has processed, then X_2 would miss both Prewrite_j and Write_j messages. Unless it is possible to bound how much transactions can be executed out of sequence at the failed site, X_2 has to send X_1 its entire commit history in order to identify the set of missing updates. (The latter approach is been proposed by several systems.)

We now show that in LAMBDA, the starting point of the missing updates can be bounded even if updates are recorded in a different order at each site. Since each message in LAMBDA NET is timestamped, the set of missing updates in X_2 can be identified using timestamp information rather than according to the ordering in X_2's log. Furthermore, site status messages clearly define the duration that a site fails. Let t_i be the timestamp of the site status message announcing site 2's failure, where site 2 holds X_2. Let t_j be the timestamp of the site status message announcing site 2's recovery. X_2 clearly has missed all the updates with timestamps between t_i and t_j. X_2 may also have missed some updates before t_i because site 2 actually fails sometimes before its failure is announced and because unprocessed messages may be lost when 2 fails, This problem, although subtle, can be solved, but the solution requires a detailed understanding of the commit process. This is explained below.

For a transaction updating logical item X, the coordinator has to obtain a response from X_2 before it can make a commit decision or it has to be notified of site 2's (X_2's) failure. In other words, if a transaction commits before t_i, the coordinator must have obtained X_2's response (X_2 therefore must have the prewrite message); if a transaction commits after t_i, the coordinator may reach a decision without X_2's response and X_2 can potentially fail before it has processed the prewrite message. To conclude, the only prewrite messages with timestamps before t_i that can be potentially missed by X_2 are those with their corresponding writing messages timestamped after t_i. The only write messages with timestamps before t_i that can be missed by X_2 are those with corresponding prewrite messages received and processed by X_2 (remember any transaction that commits before t_i must have X_2's vote!) and those transactions are still pending at X_2 (during processing the contents of prewrite messages are written to stable storage and, thus, are not affected by site failure). To obtain the proper set of missing updates, X_2 only needs to specify t_k in its request, where t_k is the smallest timestamp among the set of pending transactions at X_2.

Another difficulty in performing log-based recovery in a general network environment is to determine when to change a copy's status from recovering to operational. A recovering copy is required to obtain the most recent updates before it becomes operational. Once operational, it must continue to receive all updates on the logical data item X. Since messages can arrive out of order – a message from site B can arrive at site C before an earlier message from site A arrives – the above recovery synchronization is very difficult to achieve. Note that, in general, failure synchronization performed in a general network only requires that failure messages are synchronized among the other failure messages. Recovery synchronization actually requires that recovery status changes be synchronized with respect to other messages, such as update messages, as well. One way of simplifying the problem is to not allow transactions to commit during the recovery period, as suggested in [GOODM 83]. This increases recovery costs considerably: normal transaction execution is interrupted every time a copy recovers from failure. Other solutions execute complex protocols among the recovering copy, the copy that provides the log records, and the transaction managers of the on-going transactions to reach agreement on the copy status change.

LAMBDA provides recovery synchronization by exploiting the total ordering existing among all messages. Once site 2 recovers from failure, the active group method includes X_2 as one of the copies to receive updates on X (see previous section). That is, X_2 receives all updates on X timestamped after t_j. Updates timestamped before t_j can

be obtained from $\log(X_1)$. X_2 therefore obtains all updates that it missed. Once X_2 completes applying these updates, it changes its copy status to operational. The only externally visible effect of the copy status change is in the vote that X_2 sends back. Switching from recovering to operational does not interrupt any normal transaction processing. In fact, transaction managers of on-going transactions will not even realize that this copy status has changed. Log-based recovery therefore is considerably simplified in the LAMBDA NET environment.

3.4 Total Failure

A *total failure* occurs whenever all processes that are cooperatively executing some distributed task fail before its completion [SKEEN 83]. In a database system with replication, there are two types of total failure. A transaction total failure occurs when all processes involved in a transaction fail. A item total failure occurs when all copies of a given item fail. Since the problems in both types are similar, we will discuss only item total failure.

When a total failure of X occurs, it is important to identify the operational copy of X failing last since this copy has the most recent updates. [SKEEN 83] presents several methods for identifying this copy. The paper defines a requirement called "complete information," and shows that whenever this requirement is satisfied, the last operational copy to fail can be determined easily. On the other hand, if the requirement is not satisfied, determining the last copy is much harder.

It can be shown that if all failure messages are synchronized, the "complete information" requirement is satisfied. LAMBDA Net synchronizes failure messages; consequently, recovering from total failure becomes an easy task. This again is evidence that message synchronization simplifies the design of distributed database systems.

4. Discussion and Comparison With Other Systems

Atomic broadcast and failure detection are two basic primitives in designing distributed database systems. Atomic broadcast forms the basis of many database protocols such as transaction commit protocols, election protocols, and the "include" protocol [GOODM 83]. Failure detection is required to implement atomic broadcast. In LAMBDA NET, these are provided as network primitives. Performing these functions at the network level and providing them as system primitives not only simplifies database protocols but also better utilizes the broadcast network. (e.g., only one acknowledgement per broadcast message is required to provide atomic broadcast).

None of the systems known to the author provide atomic broadcast as a system primitive, instead they repeatedly perform a similar function at the database process level. In fact none have a true reliable broadcast facility, although many provide reliable point-to-point communication.

Failure detection is provided in some form by almost all systems. Most (e.g., SDD-1 [BERNS 80, HAMM 81] and ADAPLEX [CHAN 83a, GOODM 83]) build the failure detector at the database application level. High level timeouts are used as an indication of site failure; messages indicating (or detecting) site status are sent periodically. In contrast, LAMBDA's failure detection is performed at the network level using low level timeouts. It provides the following advantages over the failure detection mechanisms used in ADAPLEX and SDD-1.

1. It provides a more reliable failure measure. Low level timeouts are a more trustworthy indicator of failure than high level timeouts. Also, they are easier to tune to the system's status. To mistakenly declare a site failed is very expensive. In LAMBDA, the chance of a faulty declaration is reduced.
2. It provides better network utilization. Detection of site status in LAMBDA NET is obtained for free: the underlying token passing scheme detects site failures without transmitting extra messages.
3. It handles a wide range of failures: including communication failures and network partitioning. In many systems, including ADAPLEX and SDD-1, the failure detection facility cannot handle communication link failure. It assumes that if a site does not respond, it has failed.

The active group method and log-based recovery are currently being implemented in the LAMBDA database system. Contrasting LAMBDA with other distributed database system, we find that the ADAPLEX system uses similar concurrency control and crash recovery strategy. Since there does not exist a total ordering among messages transmitted, various recovery difficulties arise, as discussed in Section 3. It also implements the equivalent of atomic broadcast at least twice – for failure announcement and copy recovery – but at the database process level.

For recovery, SDD-1 provides a guaranteed delivery service for designated messages. This is achieved by sending the messages addressed to a failed site to its designated spoolers instead. The guaranteed delivery service is expensive to implement: e.g., each message requires high level acknowledgements. Although spooling does help some recovery problems, it does not help the other problems discussed herein. In particular, spooling does not ensure nonblocking commitment; consequently, SDD-1 provides a different mechanism (i.e. backup co-ordinators and a 4 phase commit protocol) to ensure this.

Tandem's system ENCOMPASS [BORR 81] focuses on providing resilient processors rather than providing reliable broadcast communication. It does not guarantee the atomic broadcast property when important messages, such as commit messages, are transmitted. When nodes fail (due to the use of dual processors, this rarely occurs), transaction processing can be blocked.

Treating replicated data is an important issue in designing distributed database systems. LAMBDA NET provides a very convenient environment to implement the read-one/write-all update strategy and the log-based recovery mechanism. This environment also simplifies other

concurrency control and crash recovery issues. Consequently, the performance of distributed database systems should improve using LAMBDA NET. This opinion is shared by [CHAN 83b], where an adapted SDD-1 concurrency control algorithm – the SDD-1 algorithm adapted for the LAMBDA NET environment – and log-based recovery have been suggested for real-time database applications.

The simplification of concurrency control and crash recovery is mainly due to the fact that LAMBDA NET provides synchronized communication. Each message carries a unique and incrementally increasing timestamp. Status information, both failure and recovery information, is not only synchronized with respect to other status information, but also synchronized with other broadcast messages. Note that only messages within one broadcast group are synchronized, this is the reason that LAMBDA treats all database sites as one large broadcast group. LAMBDA NET and the message filter process do have to process more messages. However, this function can be performed by hardware and off-loaded from the main processor. Furthermore, VLSI techniques can be used to implement these protocols in silicon.

5. Conclusions

Atomic broadcast is a powerful primitive for a distributed system. By exploiting the features of the underlying broadcast network, LAMBDA demonstrates that atomic broadcast can be implemented efficiently and inexpensively at the network level. LAMBDA NET, in addition, provides a global logical clock and a failure detector that synchronizes failure/recovery notices. The NET tolerates at least k (and normally more) benign site failures and detects network partitioning. It is failsafe: it sends "potential danger" messages to all processes and then shuts down whenever a non-tolerated benign failure occurs.

In this paper we have shown that atomic broadcast and the failure detection facility simplify transaction commitment, concurrency control, and crash recovery in a distributed database system. We have also proposed two high-level broadcast primitives, broadcast-all and broadcast-select, which are built on top of atomic broadcast. These two primitives have proven their convenience and expressiveness in the problems studied.

Broadcast networks provide a different communication environment than in point-to-point networks. In this paper, we have argued that by relegating the atomic broadcast and failure detection functions to the network, a distributed database system can be built cheaply and easily in this environment. The LAMBDA database system is currently being implemented in this environment using the protocols described herein.

6. References

[BERNS 80] P. A. Berstein, et. al, "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM TODS, March 1980.

[BERNS 81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, June 1981.

[BERNS 83] P. A. Bernstein and N. Goodman, "The Failure and Recovery Problem for Replicated Databases", ACM PODC, August 1983.

[BORR 81] A. J. Borr, "Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing", VLDB 1981.

[CHAN 83a] A. Chan, U. Dayal, S. Fox, N. Goodman, D. Ries and D. Skeen, "DDM: An ADA-Compatible Distributed Database Manager", COMPCON 83, Feb 1983.

[CHAN 83b] A. Chan, U. Dayal, S. Fox, D. Ries and J. Smith, "On Concurrency Control and Replicated Data Handling in Real-Time Distributed Database Management Systems", CCA position paper, Dec 1983.

[CHANG 83a] J. M. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols", Bell Laboratories Technical Report, Jan. 1983. Accepted for publication by ACM Transactions on Computer Systems.

[CHANG 83b] J. M. Chang, "LAMBDA: A Distributed Database System" Bell Laboratories Technical Report, March 1983, submitted for publication.

[CHANG 83c] J.M. Chang and N. F. Maxemchuk, "A Broadcast Protocol for Broadcast Networks", Proc. of Globcom, Dec 1983.

[DAVID 81] S. Davidson and H. Garcia-Molina, "Protocols for Partitioned Distributed Database Systems", TR#283, Dept of EECS, Princeton University, March 1981.

[GARCI 82] H. Garcia-Molina, "Elections in a Distributed Computing System, IEEE Transactions on Computers, Jan. 1982.

[GOODM 83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox and D. Ries, "A Recovery Algorithm for a Distributed Database System", ACM PODS, March 1983.

[GRAY 78] J. N. Gray, "Notes on Database Operating System" In Operating Systems: An advanced Course, Springer-Verlag, 1978, pp. 393-481.

[HAMME 80] M. Hammer and D. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases", ACM TODS, Dec 1980.

[LAMPO 78] L. Lamport, "Time, clocks, and the Ordering of Events in Distributed Systems", CACM, July 1978.

[LAMPO 82] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem", ACM TOPLS, July 1982.

[MAXEM 84] N. F. Maxemchuk and J. M. Chang, "Analysis of the Messages Transmitted in a Broadcast Protocol", Proc ICC 84, Amsterdam, May 1984.

[METC 76] R. M. Metcalf, D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM July 1976.

[SKEEN 81] D. Skeen, "Nonblocking Commit Protocol", Proc ACM SIGMOD, April 1981.

[SKEEN 83a] D. Skeen, "Determining the Last Process to Fail" Proc ACM PODS, March 1983.

[SKEEN 83b] D. Skeen, "Atomic Broadcasts", paper in preparation.

[SCHNE 82] F. B. Schneider, D. Gries and R. D. Schlichting, "Fast Reliable Broadcasts", Technical Report, Cornell University, Dept of Computer Science, Sept 1982.

[STONE 79] M. Stonebraker, "Concurrency Control and Consistency of Multiple copies of Data of Distributed INGRES", IEEE Trans. Software Eng. May 1979.

[WALT 82] B. Walter, "A Robust and Efficient Protocol for Checking the Availability of Remote Sites" Proc 6th Berkeley Workshop on Distributed Data Managerment and Computer Networks, Feb, 1982.

Appendix A. The Reliable Broadcast Protocol

The philosophy used in the reliable broadcast protocol [CHANG 83a, 83c] is described as follows. When there is a single transmitter, sequencing messages at all receivers is trivial: the transmitter assigns a sequence number to each message. It is not necessary for receivers to explicitly acknowledge messages since message loss is detected when a higher sequence number than expected is received. Retransmissions of lost messages can then be requested, and eventually the proper sequence of messages is obtained by all receivers. A system with multiple transmitters can be made to look like a system with a single transmitter by passing all messages through a primary receiver. Only the primary receiver is required to acknowledge a message, and it does so by assigning a sequence number (a timestamp) to each message. This replaces the single transmitter requirement. By listening to acknowledgements sent by the primary receiver, every receiver is forced to agree with the order that messages are received, and thus acknowledged, at the primary receiver. Furthermore, to avoid retaining broadcast messages indefinitely for possible retransmission and to distribute the load, primary receiver responsibility is rotated among all sites. A new primary receiver must have received all previously acknowledged messages before accepting the primary site responsibility. A site only needs to retain messages up to the last acknowledgement that the current primary receiver transmitted.

In essence, the reliable broadcast protocol uses a combination of positive acknowledgements, negative acknowledgements and token passing:

1. A source transmits a broadcast message until it receives a positive acknowledgement for the message. *One acknowledgement is sent per broadcast message.*
2. A single primary receiver, the token site, is responsible for acknowledging each message. The acknowledgements have incrementally increasing sequence numbers, called timestamps. Based on the sequence numbers, receivers request missing acknowledgements and missing broadcast messages from the token site. Note that this is a negative acknowledgement system: *no action is taken when messages are not lost.*
3. The token site responsibility is rotated among the set of operational sites in the broadcast group, called *token list,* and token passing is explicitly specified in the acknowledgement message. A site does not accept the token until it has recovered all missing messages. The token transfer message is retransmitted until the next site accepts the token by acknowledging the next broadcast message. Failures in the system are detected when the token site fails to acknowledge a broadcast message or when the next token site has failed to accept token responsibility. *The token transfer mechanism thus detects site failures.*

To guard against losing a committed message when the token site fails, an acknowledged message is not committed until k additional sites have transmitted acknowledgements. This indicates at least k+1 sites have received this message, and hence the message cannot be lost if fewer than k+1 sites fail. Resiliency therefore is obtained by introducing message delays.

The reliable broadcast protocol operates with as few as one acknowledgement per broadcast message, instead of one acknowledgement from each receiver. Extra messages, such as a special confirmation message to confirm accepting the token when there is no more broadcast message to be acknowledged, may be sent to improve system throughput. However, these messages are sent only when system utilization is low and thus do not affect performance. A detailed analysis of the number of messages transmitted in this protocol can be found in [MAXEM 84]. The protocol requires only limited storage size for message retransmission. For a broadcast group with N sites, the protocol can operate with only N-1 messages and acknowledgements retained ([CHANG 83c]).

Appendix B. The Reformation Protocol

A *communication failure* occurs when a site in the broadcast group fails to communicate with another site after R attempts. The parameter R is large enough that it is unlikely that the lack of communication is caused by lost messages alone. Therefore, when a communication failure

occurs, it indicates with high probability that the site has failed. In the reliable broadcast protocol, since the token is passed among all N operational sites in the broadcast group, a site failure will be detected within N token transfers. Whenever a failure (or recovery) is detected in the system, the *reformation protocol* [CHANG 83a] is invoked to redefine the set of operational sites and thereby create a new token list.

To preserve the correctness of the token scheme used in the reliable broadcast protocol, the reformation protocol must ensure that (1) the old token list cannot be effectively used by any site again and (2) only one new list is created. To satisfy the first requirement, the new token list must consist of one of the k sites after the old token site. Since a token site cannot commit a message without the endorsements from the k sites following it, the old token list cannot be used by any site, including a site that is not aware of the reformation process, to commit broadcast messages. This is called the *resiliency requirement*. To satisfy the second requirement, a site can join only one list and the new list must consist of a majority of sites in the broadcast group. The new list is therefore unique. This is called the *majority requirement*. To ensure that the token list can be uniquely identified by its version number, each site has a unique site # and the version number of a token list consists of (version#, site #). A site can only join a list with a higher version number than the list that it previously belonged to.

The reformation protocol is a three phase protocol initiated by the site discovering the failure. In phase I, the *initiator* invites the other sites, *the slaves*, to join its list. A new list is formed containing the responding slaves. If the new list satisfies both the resiliency and majority requirements, it is announced to all list members in phase II of the protocol. Slaves are now required to reconfirm their membership in the list. If all slaves reconfirm, then in phase III a new token is generated. Since phase III involves only the initiator and the new token site, the reformation protocol is a 3 phase protocol for the initiator and the new token site and a 2 phase protocol for all other sites. Note that if a new token is not successfully generated, the slaves and the new token site could temporarily have different views regarding the operational sites. However, since no new token is generated, no broadcast messages can be successfully transmitted and acknowledged. In this case another reformation will be initiated and all sites will eventually agree on the same set of operational sites.

Since the initiator can fail during the protocol, slaves are allowed to quit a list at any time and join a higher numbered list. In particular, they can quit between the first and second phases, and this is the reason that slaves must reconfirm their membership in phase II.

The reformation protocol ensures that when reformation completes, all sites in the new token list agree on the same set of the operational sites. The protocol also ensures that the sites in the new list have all of the acknowledged messages. If the protocol does not terminate — due to failing the majority or resiliency requirement — after several attempts, it then notifies the application program of the potentially hazardous condition. The application level can decide whether to use the new list or to block reformation until more sites recover.

The reformation protocol ensures all operational sites reaching consensus on the sites status. At the end of each reformation, a *site status message*, properly timestamped, is passed to the application program. This message consists of the new set of operational sites in the broadcast group. Since the same timestamp is used everywhere, the failure detection is synchronized with the rest of the message activity.