Distributed Logging for Transaction Processing

Dean S Daniels, Alfred Z Spector, Dean S Thompson
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Increased interest in using workstations and small processors for distributed transaction processing raises the question of how to implement the logs needed for transaction recovery Although logs can be implemented with data written to duplexed disks on each processing node, this paper argues there are advantages if log data is written to multiple *log server* nodes A simple analysis of expected logging loads leads to the conclusion that a high performance, microprocessor based processing node can support a log server if it uses efficient communication protocols and low latency, non volatile storage to buffer log data The buffer is needed to reduce the processing time per log record and to increase throughput to the logging disk An interface to the log servers using simple, robust, and efficient protocols is presented Also described are the disk data structures that the log servers use This paper concludes with a brief discussion of remaining design issues, the status of a prototype implementation, and plans for its completion

1 Introduction

Distributed transaction processing is of increasing interest, both because transactions are thought to be an important tool for building many types of distributed systems and because there are increasing transaction processing performance requirements Distributed transaction systems may comprise a collection of workstations that are widely scattered or a collection of processors that are part of a multicomputer, such as a Tandem system [Bartlett 81]

Most transaction processing systems use logging for recovery [Gray 78], and the question arises as to how each processing node should log its data This question is complex because log records may be written very frequently, there may be very large amounts of data (e g , megabytes/second), the log must be stored very reliably, and log forces must occur with little delay

In some environments, the use of shared logging facilities could have advantages in cost, performance, survivability, and operations Cost advantages can be obtained if expensive logging devices can be shared For example, in a workstation environment, it would be wasteful to dedicate duplexed disks and tapes to each workstation Performance can be improved because shared facilities can have faster hardware than could be afforded for each processing node Survivability can be better for shared facilities, because they can be specially hardened, or replicated in two or more locations Finally, there can be operational advantages because it is easier to manage high volumes of log data at a small number of logging nodes, rather than at all transaction processing nodes These benefits are accentuated in a workstation environment, but the benefits might also apply to the processing nodes of a multicomputer

This paper continues by describing the target environment for distributed logging services Section 3 presents an algorithm for replicating logs on multiple server nodes and analyzes the availability of replicated logs The design of server nodes is addressed in Section 4 The paper concludes with a description of additional design issues and the status of a prototype log server implementation

2. Target Environment

The distributed logging services described in this paper are designed for a local network of high performance microprocessor based processing nodes We anticipate processor speeds of at least a few MIPS Processing nodes might be personal workstations, or processors in a transaction processing multicomputer In either case, processing nodes may have only a single disk (or possibly no disk), but a large main memory of four to sixteen megabytes — or more Log server nodes use either high capacity disks and tapes, or write once storage such as optical disks

The network that interconnects processing nodes and servers is a high speed local area network Because processing nodes depend on being able to do logging, network failures would be disastrous Hence, the network should have multiple physical links to each processing node One way to achieve reliability is to have two complete networks, including two network interfaces in each processing node Use of the logging services described in this paper will generate a large amount of network traffic and special local networks having bandwidth greater than 10 megabits/second may be necessary in some instances

The types of transactions executed by the system will vary, depending on the nature of the processing nodes Nodes comprising a multicomputer might execute short transactions like the ET1 transaction [Anonymous et al 85] Workstation nodes might execute longer transactions on design or office automation databases These long running transactions are likely to contain many subtransactions or to use frequent save points
Unlike the parallel logging architecture proposed by Agrawal and DeWitt [Agrawal 85, Agrawal and DeWitt 85], distributed logging is intended to permit multiple transaction processors to share logging disks, rather than to accommodate a high volume of log data generated by a single (multi processor) transaction processor Of course, several distributed log servers may operate in parallel in a distributed environment that generates a large volume of log data

3 Replicated Logging

Transaction facilities typically implement recovery logs by writing data to multiple disks having independent failure modes [Gray 78] If logs are implemented using network servers, it is possible to write multiple copies of log data on separate servers, rather than on multiple disks at a single server There are at least four advantages to replicating log data using multiple servers

First, server nodes become simpler and less expensive, because they do not need multiple logging disks and controllers The number of disks and controllers that can be attached to a small server node is often limited, so this consideration can be important in practice

Second, replicating log data using multiple servers decreases the data's vulnerability to loss or corruption Separate servers are less likely to have a common failure mode than a single server with multiple disks In particular, the distance between log servers interconnected with local network technology can be much higher than the distance between disks on a single node, hence, log data replicated on multiple servers would survive many natural and man made disasters that would destroy log data replicated on a single server

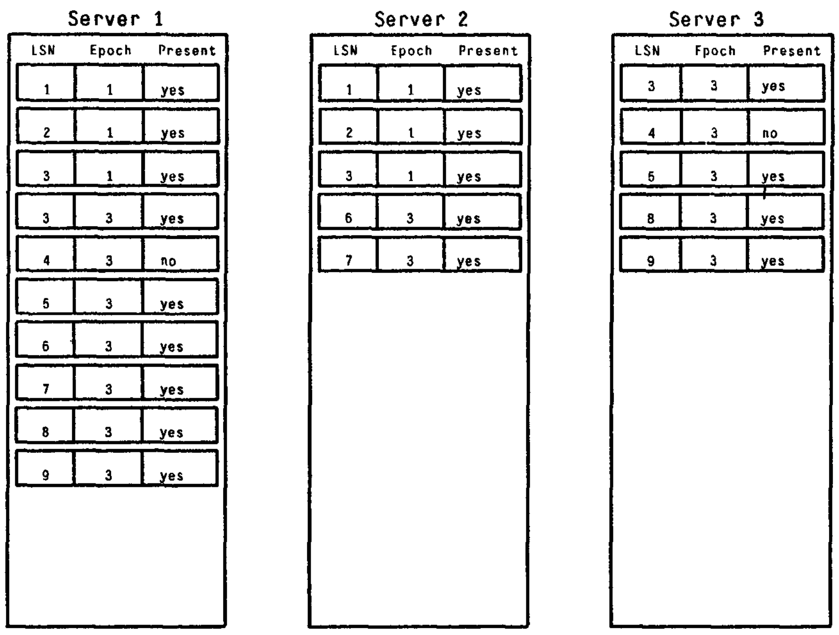
Third, replicating log data using multiple servers can increase the availability of the log both for normal transaction processing and for recovery after client node failures An analysis given in Section 3 2 shows that availability for normal processing in particular can be very greatly improved Any individual server can always be removed from the network for servicing without interrupting normal transaction processing, and in many cases without affecting client node failure recovery

Finally, the use of multiple servers rather than multiple disks on one server offers more flexibility when configuring a system Clients can choose to increase the degree of replication of their log data The opportunity also exists to trade normal processing availability for node recovery availability by varying the parameters to the replicated log algorithm described below

3 1 Replicated Log Technique

A replicated log is an instance of an abstract type that is an append only sequence of records Records in a replicated log are identified by *Log Sequence Numbers* (LSNs), which are increasing integers A replicated log is used by only one transaction processing node The use by a single client only permits a replication technique that is simpler than those that support multiple clients The data stored in a log record depends on the precise recovery and transaction management algorithms used by the client node

There are three major operations on replicated logs, though implementations would have a few others for reasons of efficiency The `WriteLog` operation takes a log record as an argument, writes it to the log, and returns the LSN associated with that record Consecutive calls to `WriteLog` return increasing LSNs The `ReadLog` operation takes an LSN as an argument and returns the corresponding log record If the argument to `ReadLog` is an LSN that has not been returned by some preceding `WriteLog` operation, an exception is signaled The `EndOfLog` operation is used to determine the LSN of the most recently written log record This replication technique could be used to


Server 1

LSN	Epoch	Present
1	1	yes
2	1	yes
3	1	yes
3	3	yes
4	3	no
5	3	yes
6	3	yes
7	3	yes
8	3	yes
9	3	yes

Server 2

LSN	Epoch	Present
1	1	yes
2	1	yes
3	1	yes
6	3	yes
7	3	yes

Server 3

LSN	Fpoch	Present
3	3	yes
4	3	no
5	3	yes
8	3	yes
9	3	yes

Figure 3-1 Three log server nodes

implement other useful log operations

The replicated log algorithm described in this section is a specialized quorum consensus algorithm [Gifford 79, Daniels and Spector 83, Bloch et al 86, Herlihy 84] that exploits the fact that a replicated log has only a single client A replicated log uses a collection of M log server nodes, with each client's log record being stored on N of the M log servers For cost reasons, log servers may store portions of the replicated logs from many clients Like the available copies replication algorithm [Bernstein and Goodman 84], this algorithm permits read operations to use only one server Concurrent access and network partitions are not concerns because a replicated log is accessed by only a single client process

`WriteLog` operations on a replicated log are implemented by sending log records to N log servers The single logging process on the client node caches information about what log servers store log records so that each `ReadLog` operation can be implemented with a request to one log server When a client node is restarted after a crash, it must initialize the cached information Naturally, the algorithm cannot require the use of an underlying transaction facility To ensure that any `WriteLog` operation that was interrupted by the client's crash is performed atomically, this replication technique uses additional fields in log records stored on log servers (as described in Section 3 1 1) and the client node initialization procedures described in Section 3 1 2

3 1 1 Log Servers and their Operations

Log servers implement an abstraction used by the replication algorithm to represent individual copies of the replicated log In addition to the log data and LSN, log records stored on log servers contain an *epoch number* and a boolean *present flag* indicating that the log record is present in the replicated directory Epochs are non decreasing integers and all log records written between two client restarts have the same epoch number If the present flag is false, no log data need be stored The present flag will be false for some log records that are written as a result of the recovery procedure performed when a client is restarted

Successive records on a log server are written with non decreasing LSNs and non decreasing epoch numbers A log record is uniquely identified by a <LSN, Epoch> pair Log servers group log records into sequences that have the same epoch number and consecutive LSNs For example, Server 1 in Figure 3 1 contains log records in the intervals (<1,1> <3,1>) and (<3,3> <9,3>)

Log servers implement three synchronous operations to support replicated logs [1] Unlike the `WriteLog` operation on replicated logs, the `ServerWriteLog` operation takes the LSN, epoch

[1] The operations are presented in a simplified way here, a more realistic interface that supports error recovery and the blocking of multiple log operations into a single server operation is described in Section 4 2

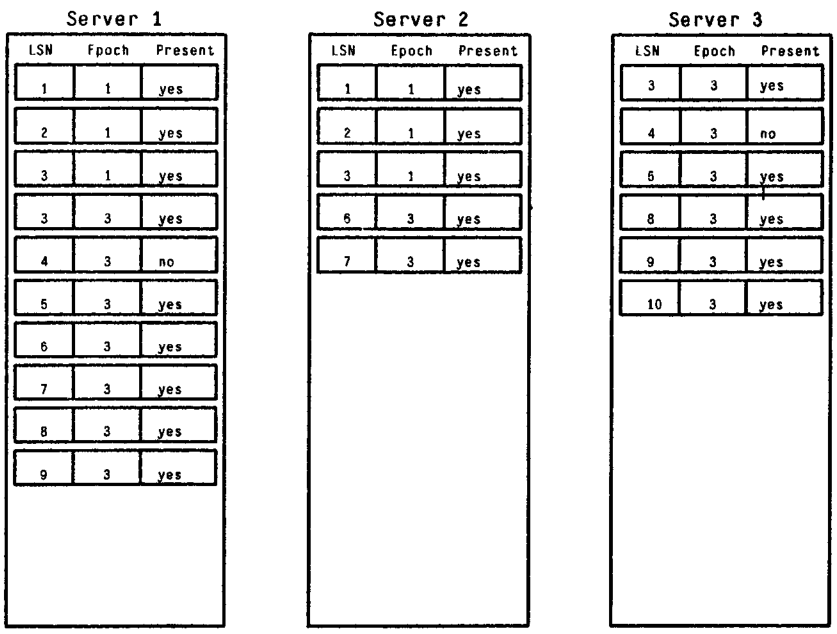


Figure 3-2 Three log server nodes with log record 10 partially written

number, and present flag for the record as arguments (along with the data) The ServerReadLog operation returns the present flag and log record with highest epoch number and the requested LSN A log server does not respond to ServerReadLog requests for records that it does not store, but it must respond to requests for records that are stored, regardless of whether they are marked present or not The IntervalList operation returns the epoch number, low LSN, and high LSN for each consecutive sequence of log records stored for a client node IntervalList is used when restarting a client node

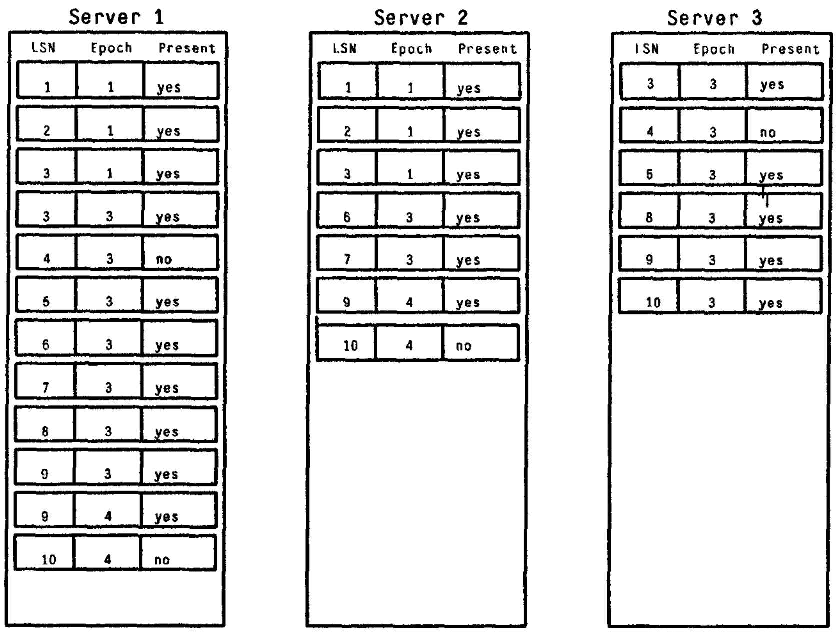
3 1 2 Replication algorithm

A replicated log is the set of <LSN,Data> pairs in all log servers such that the log record is marked present and the same LSN does not exist with a higher epoch number The replication algorithm ensures that the replicated log can be read or written despite the failure of N-1 or fewer log servers The replicated log shown in Figure 3 1 consists of records in the intervals (<1,1> <2,1>), (<3,3>), and (<5,3> <9,3>) Each of these records appears on N=2 log servers

Like other quorum consensus algorithms, the correctness of this algorithm depends on having a non empty intersection among the quorums used for different operations That is, if there are M total nodes and the client writes to N of them, with M > N, ReadLog performed with explicit voting will always require 2 or more ServerReadLog operations Yet M must be greater than N to provide high availability for WriteLog To permit ReadLog operations to be executed using a single **ServerReadLog**, this replication algorithm caches enough information on each client node to enable the client to determine which log servers store data needed for a particular ReadLog operation

Client nodes initialize their cached information when they are restarted by receiving the results of IntervalList operations from at least M - N + 1 log servers This number guarantees that a merged set of interval lists will contain at least one server storing each log record In merging the interval lists, only the entries with the highest epoch number for a particular LSN are kept In effect, this replication algorithm performs the voting needed to achieve quorum consensus for all ReadLog operations at client node initialization time That is, EndOfLog operations return the high value in the merged interval list and ReadLog operations use the list to determine a server to which to direct a ServerReadLog operation If the requested record is beyond the end of the log or if the log record returned by the ServerReadLog operation is marked not present, an exception is signaled

When a client is initialized it must also obtain a new epoch number for use with ServerWriteLog operations This epoch number must be higher than any other epoch number used during the previous operation of this client Appendix I describes a simple method for implementing an increasing unique identifier generator that can be used to assign epoch numbers and is replicated for high availability



Server 1

LSN	Epoch	Present
1	1	yes
2	1	yes
3	1	yes
3	3	yes
4	3	no
5	3	yes
6	3	yes
7	3	yes
8	3	yes
9	3	yes
9	4	yes
10	4	no

Server 2

LSN	Epoch	Present
1	1	yes
2	1	yes
3	1	yes
6	3	yes
7	3	yes
9	4	yes
10	4	no

Server 3

LSN	Epoch	Present
3	3	yes
4	3	no
6	3	yes
8	3	yes
9	3	yes
10	3	yes

Figure 3 3 Three log server nodes after execution of crash recovery procedure when Server 3 is unavailable

The `WriteLog` operation assigns an LSN by incrementing the highest LSN in the merged interval list and performs `ServerWriteLog` operations on `N` log servers If a server has received a log record in the same epoch with an LSN immediately preceding the sequence number of the new log record, it extends its current sequence of log records to include the new record, otherwise it creates a new sequence To prevent large numbers of separate sequences from being created, clients should attempt to perform consecutive writes to the same servers However, a client can switch servers when necessary

The `WriteLog` operation is not atomic and a client node crash can result in a situation where `ServerWriteLog` operations for some log record have been performed on fewer than `N` log servers In such a situation, client initialization might not gather an interval list containing the LSN for the partially written log record Figure 3 2 shows a replicated log with log record 10 partially written If Servers 1 and 2 were used for client initialization, then the transaction system recovery manager would not read log record 10 during recovery, but if Server 3 were included then record 10 would affect recovery

When a client node is initialized it is necessary to ensure that the log write that was occurring at the time of the crash appears atomic to users of the replicated log Because log writes are synchronous, there is at most one log record that has been written to fewer than `N` log servers If such a record exists, the `WriteLog` operation can not have completed and the transaction processing node cannot have depended on whether the operation was successfully or unsuccessfully performed Therefore, the log replication algorithm may report the record as existing or as not existing provided that all reports are consistent

Since there is doubt concerning only the log record with the highest LSN, it is copied from a log server storing it (using `ServerReadLog` and `ServerWriteLog`) to `N` log servers The record is copied with the client node's new epoch number Copying this log record assures that if the last record were partially written, it would subsequently appear in the interval lists of at least `N` log servers Finally, a log record marked as not present is written to `N` log servers with an LSN one higher than that of the copied record This final record has an epoch number higher than that of any copy of any partially written record and hence a partially written record with the same LSN will not be kept when interval lists are merged in any subsequent client initialization Figure 3 3 shows the replicated log from Figure 3-2 after execution of this procedure using Servers 1 and 2 [2]

[2]The careful reader will notice that log record 4 only appears as marked not present in Figures 3 1, 3 2 and 3 3 This resulted from a previous client restart using Servers 1 and 3

The client initialization procedure is not atomic. A log record can be partially copied, and the log record marked not present can be partially written. However, the procedure is restartable in that the client's recovery manager will not act on any log records prior to the completion of the recovery procedure. Once the recovery procedure completes, its effects are persistent.

3.2 The Availability of Replicated Logs

Availability is a critical issue whenever an essential resource is provided by a network server. With logs that are replicated using the previously described algorithm, there is the opportunity to trade the availabilities of different operations. In particular, `WriteLog` operations can be made more available by adding log servers, though this does decrease the availability for client node restart. Since writing log data is a much more common operation than node restart, this may be a reasonable trade off.

The availability of `ReadLog` operations depends on N and the availability of the servers storing the desired log record. The availability of the replicated log for `WriteLog` operations and client process initialization depends on both M and N and on the availability of log servers.

Assuming that log server nodes fail independently and are unavailable with probability p, then the probability that a replicated log will be available for `WriteLog` operations is simply the probability that M-N or fewer log servers are unavailable simultaneously. This probability is given by $\sum_{i=0}^{M-N}\binom{M}{i}p^i(1-p)^{M-i}$. Similarly, the probability that a replicated log will be available for client initialization[3] is given by $\sum_{i=0}^{N-1}\binom{M}{i}p^i(1-p)^{M-i}$. The probability that a replicated log will be available for reading a particular log record is $1-p^N$.

In practice, the parameter N is constrained by performance and cost considerations to having values of two or three. Thus, users of replicated logs must select values of M to provide some minimum availability for client restart. Figure 3.4 shows the expected availabilities for `WriteLog` operations and client initialization of various configurations of replicated logs under the assumption that individual servers are unavailable independently with $p=0.05$ probability.

As log servers are added (M is increased), `WriteLog` availability approaches unity very quickly. For example, consider the case of dual copy replicated logs (i.e. N=2) and M=5 log servers. For `WriteLog` operations to be unavailable in this model, at least four of the five servers must be down due to independent failures. Response to `WriteLog` operations may degrade, as fewer servers remain to carry the load, but such failures will hardly ever render `WriteLog` operations unavailable.

Client initialization availability decreases as log servers are added, because almost all servers must be available to form a quorum. In the case used as an example above, four of the five log servers must be available for client initialization. This occurs with a probability of about 0.98.

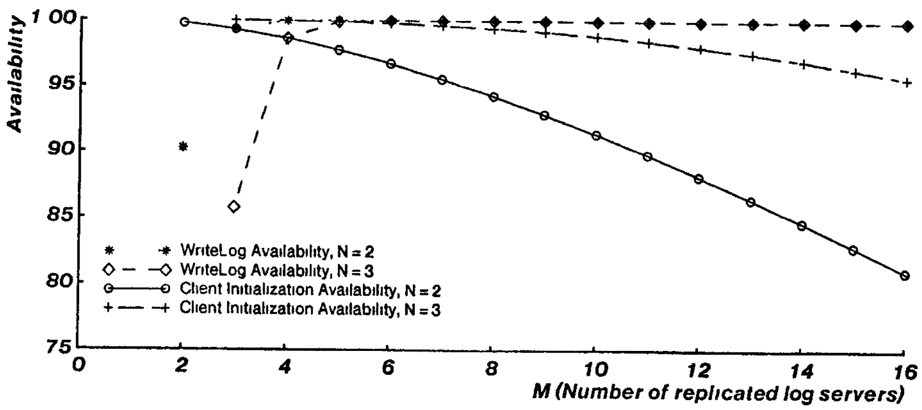


Figure 3.4 Availability of Replicated Logs with Probability of Individual Log Server Availability 0.95

[3]The probability that the replicated log will be available for client initialization also depends on the availability of the replicated increasing unique identifier generator used for obtaining a new epoch number. Appendix I gives the availability of replicated unique identifier generators. Representatives of a replicated identifier generator's state will normally be implemented on log server nodes and typical configurations will require fewer representatives than log servers for client initialization. Thus the availability of the replicated time ordered unique identifier generator does not limit the availability of replicated logs.

For applications where survivability is extremely important, or client initialization availability is extremely important, triple copy logs can be used With five log servers and triple copy replicated logs, availability for both normal processing (`WriteLog`) and client initialization is about 0 999

Certainly, replicated logs provide much higher availability than a single log server that stores multiple copies of data If only a single server were used, then `ReadLog`, `WriteLog` and client initialization would be available with probability 0 95 With dual copy replicated logs, 0 95 or better availability for client initialization would be achieved using up to $M=7$ log servers, and `WriteLog` and `ReadLog` operations would be almost always available

The above analysis predicts the instantaneous availability of replicated logs for client process initialization In practice, `M-N+1` log servers do not have to be simultaneously available to initialize a client process The client process can poll until it receives responses from enough servers to find the sites that store its log records Predicting the expected time for client process initialization to complete requires a more complicated model that includes the expected rates of log server failures and the expected times for repair

4 Log Server Design

Designing high performance servers to implement replicated logs requires determining the server hardware that is needed, the communication protocol for accessing a server, and the representation of log data Naturally, it is desirable to provide a simple design while still delivering good performance Performance is measured by the time required to perform an operation on the log server, and the number of operations per second that a log server can handle

This section begins with a simple analysis of the capacity requirements for log servers The analysis provides a concrete basis for many design decisions Next, the client interface to log servers and the protocols used for accessing them are described The last aspect of log server design addressed in detail in this paper is the representation of log data on the server Section 5 describes additional issues not treated in this paper

4 1 Log Server Capacity Requirements

The goal of this analysis is to understand what occurs in a distributed transaction processing system that is executing a heavy load In particular, it is important to determine the rate of calls to each log server, the total data volume logged in the whole system and on each log server, and the frequency with which data must be forced to disk Such an analysis also leads to an understanding of which resources become saturated first Once bottlenecks are exposed, server designs that reduce contention for these resources can be explored

For simplicity, we have chosen to analyze a load generated by a single transaction mix that we understand well The load is generated by a collection of fifty client nodes of the capacity described in Section 2 with service provided by six log servers Client nodes write log data to two log servers and are assumed to execute ten local ET1 [Anonymous et al 85] transactions per second, thus providing an aggregate rate of 500 TPS We chose this target load, not so much because we thought it would be representative of workstation loads, but because we knew that it was an ambitious load in excess of that being supported on any distributed system currently in use

Each ET1 transaction in the TABS prototype [Spector et al 85a, Spector et al 85b] transaction processing system writes 700 bytes of log data in seven log records Only the final commit record written by a local ET1 transaction must be forced to disk, preceding records are buffered in virtual memory until a force occurs or the buffer fills

If each log record were written to log servers with individual remote procedure calls (RPCs) each log server would have to process about 2400 incoming or outgoing messages per second, a load that is too high to achieve easily on moderate power processors Fortunately, recovery managers commonly support the grouping of log record writes by providing different calls for forced and buffered log writes The client interface to log servers must be designed to exploit grouping in a similar way, this permits log records to be stored on a client node until they are explicitly forced by the recovery manager

In the target load, only one log write per transaction needs to be forced to disk, thus, grouping log records until they need to be forced reduces the number of RPCs by a factor of seven Still, each server must process about 170 RPCs per second Many researchers have demonstrated that low level implementations are very important for good RPC performance [Nelson 81, Spector 82, Birrell and Nelson 84] In particular, simple, error free RPCs should be performed using only a single packet for each request and reply If this objective is met, and if the network and RPC implementation processing can be performed in one thousand instructions per packet, then communication processing will consume less than ten percent of log server CPU capacity

Fifty client nodes, each using two log servers, will generate around seven million total bits per second of network traffic With the use of multicast, this amount would be approximately halved This load could saturate many local area networks However, two networks are needed for availability reasons, and together they could support the load Higher speed, typically fiberoptic, networks having a bandwidth of approximately 100 megabits/second are becoming more prevalent [Kronenberg 86], and they could be substituted

Log servers will frequently encounter back to back requests, and so must have sophisticated network interfaces that can buffer multiple packets Client nodes will wish to send requests to different servers in parallel, and so client nodes must also be able to receive back to back (response) packets

Although message processing is not likely to be a bottleneck for log servers, performing 170 writes to non volatile storage per second could easily be a problem[4] Certainly, it is infeasible to perform seeks to different files for writes to different logs, instead records from different logs must be interleaved in a data stream that is written sequentially to disk
However, the rotational latencies would still be too high to permit each request to be forced to disk independently Waiting for requests to accumulate could delay responses, increasing lock contention and system response time Using additional disks to distribute writes increases the cost and complexity of log servers To provide acceptable logging performance with a small number of disks, log servers should have low latency, non volatile buffers so that an entire track of log data may be written to disk at once There are several technologies that may be used for constructing such a buffer, including CMOS memory with a battery for backup power Issues in designing such a buffer device are described in Section 5 1 If two thousand instructions are used to process the log records in each message and to copy them to low latency non volatile memory, and if writing a track to disk requires an additional two thousand instructions, then even with small track sizes only ten to twenty percent of a log server's CPU capacity will be used for writing log records to non volatile storage Disk utilization will be higher close to fifty percent for slow disks with small tracks The additional disk utilization induced by read operations is difficult to predict because it depends on the frequency of reads and on whether reads are mainly for sequential log records The use of local caches for processing transaction aborts, as described in Section 5 2, means that server read operations will be used mainly for node restart and media recovery processing

With this target load, approximately ten billion bytes of log data will be written to each log server per day Current technology permits the storage of this much data online, so that simple log space management strategies could be used For example, database dumps could be taken daily, and the online log could simply accumulate between dumps However, storage for this much log data would dominate log server hardware costs It may be desirable for cost, performance, or reliability reasons to used more sophisticated checkpointing and dumping strategies to limit the online log needed for node or media recovery as Section 5 3 discusses

To summarize, this capacity analysis has exposed several important points for log server design First, clients must access log servers through interfaces which group together multiple log records and send them using specialized low level protocols Second, multiple high bandwidth networks must be used and network interfaces must be capable of receiving back to back packets Third, log records from multiple clients must be merged into a single data stream that is buffered on a special low latency non volatile storage device and then written to disk a track at a time Finally, the volume of log data generated, while large, is not so great as to require complicated processing simply to reduce online log storage requirements If these points are taken into account, it should be possible to achieve acceptable performance from servers implemented using the same processors as client nodes

4 2 Log Server Interfaces and Protocols

The preceding section demonstrated that the interfaces and protocols used to access log servers must be carefully designed to avoid bottlenecks First, the interface must transfer multiple log records in each network message to reduce the number of messages that must be processed Second, the interface must be implemented using specialized protocols, rather than being layered on top of expensive general purpose protocols Additionally, the protocols used for accessing log servers should permit large volumes of data to be written to the server efficiently

[4]Because power failures are likely to be a common failure mode for log servers, it is not acceptable to buffer log data in volatile storage

Asynchronous Messages from Client to Log Server

```
WriteLog(IN ClientId, EpochNum, LSNs, LogRecords)

ForceLog(IN ClientId, EpochNum, LSNs, LogRecords)

NewInterval(IN ClientId, EpochNum, StartingLSN)
```

Asynchronous Messages from Log Server to Client

```
NewHighLSN(IN NewHighLSN)

MissingInterval(IN MissingInterval)
```

Synchronous Calls from Client to Log Server

```
IntervalList(IN ClientId, OUT IntervalList)

ReadLogForward(IN ClientId, LSN, OUT  LSNs, LogRecords, PresentFlags)

ReadLogBackward(IN ClientId, LSN, OUT  LSNs, LogRecords, PresentFlags)

CopyLog(IN ClientId, EpochNum, LSNs, LogRecords, PresentFlags)

InstallCopies(IN ClientId, EpochNum)
```

Figure 4-1 Client Interface to Log Servers

The need for streaming is not well illustrated by ET1 transactions, but it is important for transactions that update larger amounts of data

Extra care must be taken to make the low level protocols robust Lost or delayed messages must be tolerated, and recovery from server failures must be possible Realistically, remote logging will only be feasible when local area networks are used, hence, the protocols should exploit the inherent reliability of local area networks and use end to end error detection and correction [Saltzer et al 84] to eliminate the expense of redundant acknowledgments and error detection

Simplicity, both in specification and implementation, is an additional desirable property for the interface to a log server RPCs possess this property and they can be implemented very efficiently Unfortunately, RPCs are inherently synchronous and do not permit the eff cient streaming of large amounts of data The interface presented here includes strict RPCs for infrequently used operations, such as for reading log records, and asynchronous messages for writing and acknowledging log records

Figure 4 1 shows RPC and message header definitions for the log server and client interface This interface differs from the abstract operations defined on log servers in Section 3 1 because of the support for transmission of multiple log records in each call, streaming of data to log servers, and error detection and recovery

To establish communication with a log server, a client initiates a three way handshake Both client and server then maintain a small amount of state while the connection is active This allows packets to contain permanently unique sequence numbers, and permits duplicate packets to be detected even across a crash of the receiving node All calls participate in a moving window flow control strategy at the packet level An allocation inserted in every packet specifies the highest sequence number the other party is permitted to send without waiting Deadlocks are prevented by allowing either party to exceed its allocation, so long as it pauses several seconds between packets to avoid overrunning the receiver Each party attempts to supply the other with unused allocation at all times This connection establishment and flow control mechanism is based on a tutorial by Watson [Watson 81]

The `WriteLog` and `ForceLog` messages, and the `ReadLogForward`, and `ReadLogBackward` calls transmit multiple log records Client processes and log servers attempt to pack as many log records as will fit in a network packet in each call The `ReadLogForward` and `ReadLogBackward` calls differ as to whether log records with log sequence number (LSN) greater or less than the input LSN are used to fill the packet The `IntervalList` call is used when a client is initialized as described in Section 3 1 The input parameter to this call is a client identifier and the output is a list of intervals of log records stored by the server for that client

Multiple `WriteLog` operations on a distributed log are grouped together by the client and records are streamed to log servers asynchronously Error detection and recovery is based on requested positive acknowledgments, prompt negative acknowledgments, and prompt responses to negative acknowledgments A client writes log records with the `ForceLog` message when it needs an immediate acknowledgment, and with the `WriteLog` message when it does not If it uses the `ForceLog` message and does not get a response, it retries a number of times before moving to a different server A log server acknowledges a successful `ForceLog` message by returning the highest forced log sequence number in a `NewHighLSN` message A server detects lost messages when it receives a `ForceLog` or `WriteLog` message with log sequence numbers that are not contiguous with those it has previously received from the same client It notifies the client of the missing interval immediately through a `MissingInterval` message

When a client receives a `MissingInterval` message it will either resend the missing log records in a `ForceLog` message, or use the `NewInterval` message to inform the server that it should ignore the missing log records and start a new interval The `NewInterval` message is used when the missing log records have already been written to other log servers, as happens when a client switches log servers

Log servers should make every effort to reply to `IntervalList`, `ReadLogForward` and `ReadLogBackward` calls, but they are free to ignore `ForceLog` and `WriteLog` messages if they become too heavily loaded Clients will simply assume that the server has failed and will take their logging elsewhere

Because the client interface to log servers is designed to group log records and send multiple packets of log records asynchronously, more than one record might have been written to fewer than N log servers when a client node crashes The client must limit the number of records contained in unacknowledged `WriteLog` and `ForceLog` messages to ensure that no more than δ log records are partially written Even with such a bound, two additional calls on log servers are need for client recovery The `CopyLog` synchronous call is used to rewrite records that may have been partially written Unlike the `ForceLog` and `WriteLog` messages, log servers accept `CopyLog` calls for records with LSNs that are lower than the highest log sequence number written to the log server The `InstallCopies` call is used to atomically install all log records copied (with the `CopyLog` call) with a particular epoch number Thus, during recovery the most recent δ log records are copied with a new epoch number using `CopyLog` calls Then, δ new log records marked not present are written with `CopyLog` calls Finally, the copied records are installed with the `InstallCopies` call

The protocol described here is not the only one that might be used If most log records are smaller than a network packet, the log sequence numbers themselves can be used efficiently for duplicate detection and flow control This permits a much simpler implementation (and possibly better performance), and eliminates the need for special messages to establish a connection The disadvantage to this approach is that any log record larger than a network packet must be sent with an acknowledgment for each packet, or through a separate connection based protocol

If most log data must be written synchronously (as will be the case when transactions are short), it is appropriate to use a more synchronous interface than that described here A strictly synchronous interface will be simpler to implement, and might perform better if the need for asynchronous streaming of large amounts of data is rare

4 3 Disk Data Structures for Log Servers

The capacity analysis in Section 4 1 demonstrated the need for data structures that use log servers' disks efficiently In particular, log servers should not have to perform disk seeks in order to log data from different clients Thus, the first objective in the design of the disk representation for log servers is to reduce seeks while writing An additional objective is to design data structures that permit the use of write once (optical) storage, because that technology may prove useful for capacity and reliability reasons

There are two types of data that must be kept in non volatile storage by log servers First, the server must store the interval lists describing the consecutive sequences of log records stored for each client node Second, the server must store the consecutive sequences of log records themselves

An essential assumption of the replicated logging algorithm is that interval lists are short In general, new intervals should be created only when a client crashes or when a server crashes or runs out of capacity Storing one interval requires space for three integers the epoch number and a beginning and ending LSN This will be only a tiny fraction of the space required to store the log records in the interval

Because interval lists are short, it is reasonable for a server to keep them in volatile memory during normal operation This is particularly convenient because the last intervals in the lists for active clients are changing as new log records are written Interval lists are checkpointed to non volatile storage periodically They may be checkpointed to a known location on a reusable disk or to a write once disk along with the log data stream After a crash, a server must scan the end of the log data stream to find the ends of active intervals, unless there is sufficient low latency non volatile memory to store active intervals

Organization of the storage for the consecutive sequences of log records is a more difficult problem To reduce seeking, data from different clients must be interleaved on disk Any sequence may be hundreds of megabytes in length and be spread over gigabytes of disk A data structure that permits random access by log sequence number is needed

Logarithmic read access to records may be achieved using a data structure that we call an *append-forest* New records may be added to an append forest in constant time using append only storage, providing that keys are appended to the tree in strictly increasing order A complete append forest (with 2^n-1 nodes) resembles and is accessed in the same manner as a binary search tree having the following two properties

1 The key of the root of any subtree is greater than all its descendants' keys

2 All the keys in the right subtree of any node are greater than all keys in the left subtree of the node

An incomplete append forest (with more than 2^n-1 nodes and less than $2^{n+1}-1$ nodes) consists of a forest of (at most $n+1$) complete binary search trees, as described above Trees in the forest have height n or less, and only the two smallest trees may have the same height All but at most one of the smallest trees in the forest will be left subtrees of nodes in a complete append forest (with $2^{n+1}-1$ nodes) Figure 4 2 is a schematic diagram of an append forest

All nodes in the append forest are reachable from its root, which is the node most recently appended to the forest The data structure supports incomplete append forests by adding an extra pointer, called the forest pointer, to each node This pointer links a new append forest root (which is also the root of the right most tree in the append forest) to the root of the next tree to the left A chain of forest pointers pointers permits access to nodes that are not dependents of the root of the append forest

Searches in an append forest follow a chain of forest pointers from the root until a tree (potentially) containing the desired key is found Binary tree search is then used on the tree An append forest with n nodes contains at most$\lceil log_2(n)\rceil$ trees in its forest, therefore searches perform $O(log_2(n))$ pointer traversals

Figure 4-3 illustrates an eleven node append forest The solid lines are the pointers for the right and left sons of the trees in the forest Dashed lines are the forest pointers that would be used for searches on the eleven node append forest Dotted lines are forest pointers that were used for searches in smaller trees The last node inserted into the append forest was the node with key 11 A new root with key 12 would be appended with a forest

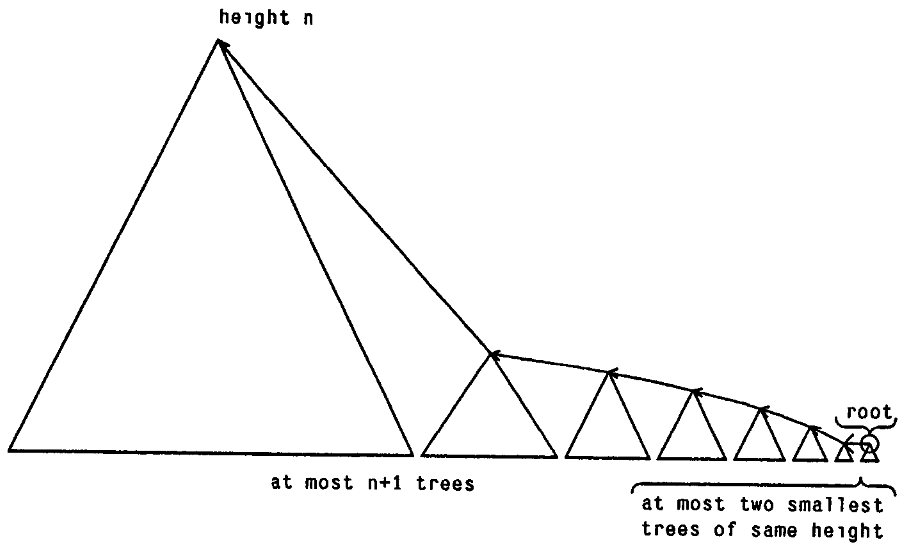


Figure 4-2 Schematic of incomplete append forest

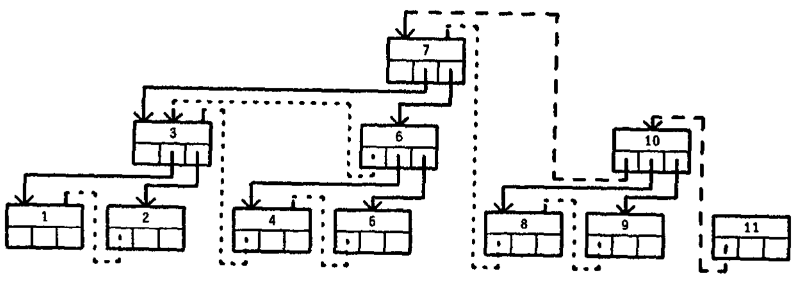


Figure 4-3 Eleven Node Append forest

pointer linking it to the node with key 11 An additional node with key 13 would have height 1, the nodes with keys 11 and 12 as its left and right sons, and a forest pointer linking it to the tree rooted at the node with key 10 Another node with key 14 could then be added with the nodes with keys 10 and 13 as sons, and a forest pointer pointing to the node with key 7

When an append forest is used to index a log server client's records, the keys will be ranges of log sequence numbers Each node of the append forest will contain pointers to each log record in its range With this data structure, each page sized node of the tree can index one thousand or more records

5 Discussion

This paper has presented an approach to distributed logging for transaction processing The approach is intended to support transaction processing for workstations or to be used to construct multicomputer systems for high performance transaction processing Designs for a replicated logging algorithm, for an interface to log servers, and for disk data structures for log storage have been presented Additional design issues are described below, and the paper concludes with the status of our implementation of prototype log servers

5 1 Low Latency Non volatile Memory

CMOS battery backup memory is almost certainly the technology of choice for implementing low latency non volatile memory Less certain is the mechanism by which the host processor should access the memory Section 4 3 indicated than an additional use for low latency non volatile memory besides disk buffering would be storing active intervals It is likely that others could be found and, therefore it can be argued that the non volatile memory should be accessible as ordinary memory, rather than integrated into a disk controller However, data in directly addressable non volatile memory may be more prone to corruption by software error Needham et al [Needham et al 83] have suggested that a solution to this problem is to provide hardware to help check that each new value for the non volatile memory was computed from a previous value

5 2 Log Record Splitting and Caching

Often, log records written by a recovery manager contain independent redo and undo components The redo component of a log record must be written stably to the log before transaction commit The undo component of a log record does not need to be written to the log until just before the pages referenced in the log record are written to non volatile storage Frequently transactions commit before the pages they modify are written to non volatile storage

The volume of logged data may be reduced if log records can be *split* into separate redo and undo components Redo components of log records are sent to log servers as they are generated, with the rest of the log data stream Undo components of log records are *cached* in virtual memory at client nodes When a transaction commits, the undo components of log records written by the transaction are flushed from the cache If a page referenced by an undo component of a log record in the cache is scheduled for cleaning, the undo component must be sent to log servers first If a transaction aborts while the undo components of its log records are in the cache, then the log records are available locally and do not need to be retrieved from a log server

The performance improvements possible with log record splitting and caching depend on the size of the cache, and on the length of transactions If transactions are very short, then the fraction of log records that may be split will be small, and splitting will not save much data volume Very long running transactions will not complete before pages they modify are cleaned, and splitting will also not save data volume The cached log records will speed up aborts and relieve disk arm movement contention on log servers because log reads will go to the caches at the clients

5 3 Log Space Management

Log servers may be required to store large amounts of data as indicated in Section 4 1 Various space management functions may be used in different combinations to reduce the amount of online log data There are at least four functions that can be combined to develop a space management strategy First, client recovery managers can use checkpoints and other mechanisms to limit the online log storage required for node recovery Second, periodic dumps can be used to limit the total amount of log data needed for media failure recovery Third, log data can be spooled to offline storage Finally, log data can be compressed to eliminate redundant or unnecessary log records

Log space management strategies should be compared in terms of their cost and performance for various recovery operations Recovery operations of interest include node recovery, media failure recovery, the repair of a log when one redundant copy is lost, and recovery from a disaster that destroys all copies of part of a log Relevant cost measures include online storage space, processing requirements in log servers and clients, and software complexity Performance measures include whether recovery operations can be performed with online or offline data and the relative speed of the operations

5 4 Load Assignment

The interface presented in Section 4 2 does not contain protocols for assigning clients to log servers Ideally, clients should distribute their load evenly among log servers so as to minimize response times

If the only technique for detecting overloaded servers is for a client to recognize degraded performance with a short timeout, then clients might change servers too frequently resulting in very long interval lists If servers shed load by ignoring clients, then clients of failed servers might try one server after another without success Presumably, simple decentralized strategies for assigning loads fairly can be used The development of these strategies is likely to be a problem that is very amenable to analytic modeling and simple experimentation

5 5 Common Commit Coordination

If remote logging were performed using a server having mirrored disks, rather than using the replicated logging algorithm described in Section 3, that server could be a coordinator for an optimized commit protocol The number of messages and the number of forces of data to non volatile storage required for commit could be reduced, compared with frequently used distributed commit protocols [Lindsay et al 79] Optimizations are applicable only when transactions modify data on more than one node Latency for log forces is not as great a consideration if low latency non-volatile storage is being used in log servers, and careful design of commit protocols can reduce the number of messages required for commit of distributed transactions Still, if multi node transactions are frequent then common commit coordination is an argument against replicated logging

5 6 Implementing A Prototype Log Server

The authors are currently implementing a prototype distributed logging service based on the design presented in this paper The implementation is being carried out in several stages

The first stage, an implementation of the log server interface and protocols described in Section 4 2 is complete It uses Perq workstations running the Accent operating system [Perq Systems Corporation 84, Rashid and Robertson 81] Accent inter node communication is not as low level or efficient as Section 4 1 suggests is necessary, and only a skeleton log server process without data storage was implemented This implementation was completed in January 1986 and was used for validating the protocols

The second stage of implementation augmented the first stage with data storage in the log server's virtual memory The TABS distributed transaction processing system was used as a client for this stage of the log server Thus, the second stage permitted operational experience with a working transaction system as a client As of April 1986, remote logging to virtual memory on two remote servers used less than twice the elapsed time required for local logging to a single disk

The first version of our final implementation of distributed logging began operating in February 1987 in support of the Camelot distributed transaction facility [Spector et al 86] Both Camelot and the log servers use the Mach operating system [Accetta et al 86] and currently run on both IBM RT PCs and DEC Vaxes This implementation uses low level protocol implementations and stores log data on magnetic disks The append forest is not currently implemented because the amount of data stored is comparatively small When optical disks are available, the append forest will be used Low latency non volatile storage is assumed by the implementation and we expect to implement such storage with a battery based standby power supply for the entire workstation After performance tuning and evaluation, this implementation will be the basis for additional research in the other aspects of distributed logging mentioned above

Acknowledgments

We wish to thank Sherri Menees, Randy Pausch, Bruce Lindsay, and an anonymous referee for reading and commenting on this paper Jim Gray also contributed to this work through a lively discussion

I Replicated Increasing Unique Identifiers

This appendix considers the problem of generating increasing unique identifiers Section 3 1 describes how these identifiers are used to distinguish log records written in different client crash epochs and hence to make atomic any log writes that might be interrupted by crashes This method for replicating identifier generators only permits a a single client process to generate identifiers at one time Atomic updates of data at more than one node are not required for this method of replicating unique identifier generators

Increasing identifiers are given out by a replicated abstract datatype called a *replicated identifier generator* The only operation provided by a replicated identifier generator is NewID, a function that returns a new unique identifier Identifiers issued by the same generator can be compared with = (equal) and < (less than) operators Two identifiers are equal only if they are the result of the same NewID invocation One identifier is less than another only if it was the result of an earlier invocation of NewID

Identifiers given out by a replicated identifier generator are integers and integer comparisons are used for the < and = operations The state of the replicated identifier generator is replicated on N *generator state representative* nodes that each store an integer in non volatile storage[5] Generator state representatives provide Read and Write operations that are atomic at individual representatives

The NewID operation first reads the generator state from $\lceil\frac{N+1}{2}\rceil$ representatives Then, NewID writes a value higher than any read to $\lceil\frac{N}{2}\rceil$ representatives Any overlapping assignment of reads and writes can be used Finally, the value written is returned as a new identifier

Because the set of generator state representatives read by any NewID operation intersects the set of representatives written by all preceding NewID operations that returned values, identifiers returned by a NewID invocation are always greater than those returned by previous invocations If a crash interrupts a NewID operation, then a value written to too few representatives could be omitted from the sequence of identifiers generated

The difference between an infinite sequence of unique identifiers generated as described here and a sequence of log sequence numbers generated by the replicated log WriteLog operation described in Section 3 1 is that the replicated log ReadLog operation may be used to determine whether the sequence contains some integer Similarly, the scheme described here provides no way to determine the last identifier generated

The availability of the replicated increasing unique identifier generator depends on the availability of the generator state representatives and on the number of generator state representatives If generator state representatives are unavailable with probability p then the probability that a replicated unique identifier generator is available is the probability that $\lfloor\frac{N-1}{2}\rfloor$ or fewer nodes are unavailable simultaneously This probability is given by $\sum_{i=0}^{\lfloor\frac{N-1}{2}\rfloor}\binom{N}{i}p^i(1-p)^{N-i}$

[5]Append only storage may be used to implement generator state representatives

References

[Accetta et al 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young Mach A New Kernel Foundation for UNIX Development In *Proceedings of Summer Usenix* July, 1986

[Agrawal 85] Rakesh Agrawal A Parallel Logging Algorithm for Multiprocessor Database Machines In *Proceedings of the Fourth International Workshop on Database Machines*, pages 256 276 March, 1985

[Agrawal and DeWitt 85] Rakesh Agrawal, David J DeWitt Recovery Architectures for Multiprocessor Database Machines In *Proceedings of ACM SIGMOD 1985 International Conference on Management of Data*, pages 132 145 May, 1985

[Anonymous et al 85] Anonymous, et al A Measure of Transaction Processing Power *Datamation* 31(7), April, 1985 Also available as Technical Report TR 85 2, Tandem Corporation, Cupertino, California, January 1985

[Bartlett 81] Joel Bartlett A NonStop™ Kernel In *Proceedings of the Eighth Symposium on Operating System Principles* ACM, 1981

[Bernstein and Goodman 84] P Bernstein and N Goodman An algorithm for concurrency control and recovery in replicated distributed databases *ACM Transactions on Database Systems* 9(4) 596 615, December, 1984

[Birrell and Nelson 84] Andrew D Birrell, Bruce J Nelson Implementing Remote Procedure Calls *ACM Transactions on Computer Systems* 2(1) 39 59, February, 1984

[Bloch et al 86] Joshua J Bloch, Dean S Daniels, Alfred Z Spector *A Weighted Voting Algorithm for Replicated Directories* Technical Report CMU CS 86 132, Carnegie Mellon University, June, 1986 Revision of Report CMU CS 84-114, April 1984 To appear in JACM in 1987

[Daniels and Spector 83] Dean S Daniels, Alfred Z Spector An Algorithm for Replicated Directories In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 104 113 ACM, August, 1983 Also available in Operating Systems Review 20(1), January 1986, pp 24 43

[Gifford 79] David K Gifford Weighted Voting for Replicated Data In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 150 162 ACM, December, 1979

[Gray 78] James N Gray Notes on Database Operating Systems In R Bayer, R M Graham, G Seegmuller (editors), *Lecture Notes in Computer Science* Volume 60 *Operating Systems - An Advanced Course*, pages 393 481 Springer Verlag, 1978 Also available as Technical Report RJ2188, IBM Research Laboratory, San Jose, California, 1978

[Herlihy 84] Maurice P Herlihy *General Quorum Consensus A Replication Method for Abstract Data Types* Technical Report CMU CS 84 164, Carnegie Mellon University, December, 1984

[Kronenberg 86] Nancy P Kronenberg, Henry M Levy, and William D Strecker VAXclusters A Closely Coupled Distributed System *ACM Transactions on Computer Systems* 4(2), May, 1986 Presented at the Tenth Symposium on Operating System Principles, Orcas Island, Washington, December, 1985

[Lindsay et al 79] Bruce G Lindsay, et al *Notes on Distributed Databases* Technical Report RJ2571, IBM Research Laboratory, San Jose, California, July, 1979 Also appears in Droffen and Poole (editors), *Distributed Databases*, Cambridge University Press, 1980

[Needham et al 83] R M Needham, A J Herbert, J G Mitchell How to Connect Stable Memory to a Computer *Operating Systems Review* 17(1) 16, January, 1983

[Nelson 81] Bruce Jay Nelson *Remote Procedure Call* PhD thesis, Carnegie Mellon University, May, 1981 Available as Technical Report CMU CS 81 119a, Carnegie Mellon University

[Perq Systems Corporation 84] *Perq System Overview* March 1984 edition, Perq Systems Corporation, Pittsburgh, Pennsylvania, 1984

[Rashid and Robertson 81] Richard Rashid, George Robertson Accent A Communication Oriented Network Operating System Kernel In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 64 75 ACM, December, 1981

[Saltzer et al 84] J H Saltzer, D P Reed, D D Clark End To End Arguments in System Design *ACM Transactions on Computer Systems* 2(4) 277 288, November, 1984

[Spector 82] Alfred Z Spector Performing Remote Operations Efficiently on a Local Computer Network *Communications of the ACM* 25(4) 246 260, April, 1982

[Spector et al 85a] Alfred Z Spector, Jacob Butcher, Dean S Daniels, Daniel J Duchamp, Jeffrey L Eppinger, Charles E Fineman, Abdelsalam Heddaya, Peter M Schwarz Support for Distributed Transactions in the TABS Prototype *IEEE Transactions on Software Engineering* SE 11(6) 520 530, June, 1985 Also available in Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Springs, Maryland, IEEE, October, 1984 and as Technical Report CMU CS 84-132, Carnegie Mellon University, July, 1984

[Spector et al 85b] Alfred Z Spector, Dean S Daniels, Daniel J Duchamp, Jeffrey L Eppinger, Randy Pausch Distributed Transactions for Reliable Systems In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127 146 ACM, December, 1985 Also available in *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company, New York, and as Technical Report CMU CS 85 117, Carnegie Mellon University, September 1985

[Spector et al 86] Alfred Z Spector, Joshua J Bloch, Dean S Daniels, Richard P Draves, Dan Duchamp, Jeffrey L Eppinger, Sherri G Menees, Dean S Thompson The Camelot Project *Database Engineering* 9(4), December, 1986 Also available as Technical Report CMU CS-86 166, Carnegie Mellon University, November 1986

[Watson 81] R W Watson IPC Interface and End To End Protocols In B W Lampson (editors), *Lecture Notes in Computer Science* Volume 105 *Distributed Systems - Architecture and Implementation An Advanced Course*, chapter 7, pages 140 174 Springer Verlag, 1981