The Raid Distributed Database System

BHARAT BHARGAVA, SENIOR MEMBER, IEEE, AND JOHN RIEDL

Abstract—**Raid is a robust and adaptable distributed database system for transaction processing. Raid is a message-passing system, with server processes on each site. The servers manage concurrent processing, consistent replicated copies during site failures, and atomic distributed commitment. A high-level, layered communications package provides a clean, location independent interface between servers. The latest design of the communications package delivers messages via shared memory in a high-performance configuration in which several servers are linked into a single process. Raid provides the infrastructure to experimentally investigate various methods for supporting reliable distributed transaction processing. Measurements on transaction processing time and server CPU time are presented. Data and conclusions of experiments in three categories are also presented: communications software, consistent replicated copy control during site failures, and concurrent distributed checkpointing. A software tool for the evaluation of the implementation of transaction processing algorithms in an operating system kernel is proposed.**

Index Terms—**Communications, database, distributed systems, reliability, replication, transaction processing.**

I. Introduction

RAID stands for robust and adaptable distributed database system. We discuss the design and implementation of Raid. We include measurements that we have performed, and our ongoing experimental work. We provide insight into the theoretical underpinnings of adaptability [10], and discuss briefly the ways in which the Raid design provides an infrastructure for adaptability. We discussed design principles and experiences with them in different sections. In the last section, we present a retrospection on what we have learned from our experience with Raid, and on the future research directions that have arisen from this experience.

A. Implementation Objectives

Raid has been implemented to obtain measurements and conduct experiments that provide empirical evaluation of the algorithms used in distributed database systems. This research helps in identifying principles and design alternatives that contribute towards reliable, high-performance transaction processing. Three classes of experiments are supported by the Raid project: simulation experiments, microexperiments, and macroexperiments. Each experiment requires four steps: design and setup, measurements and observations, analysis of the data, and conclusions indicating the relevance to reliable, high performance transaction processing.

The simulation experiments are run on subsystems called *mini-Raid* [9] and *Seth* [14]. These systems are smaller versions of Raid. Mini-Raid is a simulation environment within which new ideas can be tested before they are implemented in Raid itself. Seth supports evaluation of quorum consensus-based protocols. Microexperiments are run to measure the performance of a particular subsystem of Raid. The performance figures for the Raid communications package were obtained with microexperiments. The results of microexperiments are used to provide input parameter values for the simulation experiments and to predict the performance of alternative configurations of Raid. Macroexperiments measure the transaction processing performance of the Raid system in traditional terms, such as throughput or delay. These are done on the full Raid system, using both simulated transactions and transaction benchmarks [1], [12].

Some of the topics on which experiments have already been performed using Raid are: replicated copy control during site failure and recovery, measurements and enhancements of the communication facilities, and distributed checkpointing algorithms. These experiments are described in Section III.

Another objective of our research is to experiment with the implementation of an adaptable system [10] that can reconfigure based on performance and reliability requirements. Many algorithms for concurrency control, network partitioning, replication control, and commitment have been proposed, analyzed and evaluated in the literature [3]. However, a complete system requires a unique combination of these algorithms to satisfy a given application. Due to failures in a distributed system, graceful reconfiguration is required to continue transaction processing. Reconfiguration is also needed when performance and reliability characteristics of the current set of transaction processing algorithms are unacceptable for the current application load. To allow easy reconfiguration, we need to design systems that are adaptable. Adaptability [10] requires that algorithms and their respective data structures be designed to support switching to new configurations.

Our approach for reconfiguration and adaptability is based on the definition of a *sequencer* [10]. A sequencer is an abstraction of a control component of the distributed system, such as the network partitioning controller, the atomicity controller, or the concurrency controller. Sequencers accept actions as input, and schedule the actions

Manuscript received December 30, 1987; revised February 1, 1989. This work was supported by NASA and AIRMICS under Grant NAG-1-676, by UNISYS Corporation, and by a David Ross fellowship.

The authors are with the Department of Computer Sciences, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 8927384.

0098-5589/89/0600-0726$01.00 © 1989 IEEE

for execution. Each sequencer maintains state information, such as the number of votes assigned to each site, or the location of tokens for data items. The crucial problem in reconfiguring or adapting between sequencers is preserving the correctness of the state information. We have three methods for adapting between sequencers [10]. The *Global State* method maintains a single representation of the sequencer state information. All sequencer implementations can share this state information. The *Converting State* method is based on converting sequencer state information from one representation to another as needed. The *Suffix-Sufficient State* method works by introducing an intermediate stage during which preconditions for the new algorithm are satisfied. This method has the advantage of being able to work with arbitrary sequencers without knowledge of the internal state maintained by the sequencer. More details are given in·[10].

The design of Raid is flexible and modular, providing an infrastructure for switching algorithms. Each subsystem is implemented as a separate server, communicating with other servers via a clearly defined message protocol, so that a new implementation can be easily substituted for an existing implementation. Furthermore, we are modifying the concurrency controller to dynamically change between different methods while Raid is processing transactions. We use the suffix-sufficient state method of conversion. The new concurrency controller runs in parallel with the old concurrency controller, until the new one has absorbed enough state to take over serializing transactions. When the new concurrency controller takes over, the history information that is used for serialization is truncated at the point that the new concurrency controller was started. Uncommitted transactions that were active before this point must be aborted. Choosing which method should be running in a given situation is a complex problem, so we have implemented a prototype expert system to allow Raid to be truly adaptive, in the sense that it can automatically change itself to conform to its environment [11]. This expert system uses a database of rules to produce an estimate of the optimum concurrency controller for a given transaction mix, along with a belief value in its reasoning process.

B. Distinguishing Features

The Raid system is implemented in 20K lines of C code, and can run on either Vaxen or Suns under 4.3 BSD UNIX.[1] Raid divides the functions of transaction processing into software modules called *servers*. An operating system process can implement the capabilities of a single server or of a collection of servers. The server design has facilitated the implementation effort by providing for flexibility, and by explicitly defining the interfaces between servers. This provides the infrastructure for adaptability. Unless concurrency and parallelism can be provided to keep all servers busy, such a design has performance problems, both because of the high cost of communicating between processes and because of the increased operating system overhead from context switching between multiple processes. An alternate design for an operational version of Raid requires multiple servers to reside in the same UNIX process, and communicate via shared memory. The following are special features of Raid:

- Raid is designed as a modular, message-passing system to support easy extensions and modifications. Servers can be relocated, and new implementations of servers can be substituted.
- We are completing the implementation of dynamic adaptability for the Raid concurrency controller. The concurrency controller presently chooses statically between four concurrency control algorithms. Soon it will be able to dynamically switch, without suspending transaction processing.
- The Raid communication subsystem provides location transparent addressing, and supports multiple virtual sites on a single physical host.
- Raid has facilities for replicated copy management that can handle site failures. We are in the process of implementing a feature that will automatically refresh outdated data copies after recovery.
- RaidTool is a window-based interface, designed to be a front-end to the Raid system. RaidTool permits an operator to configure a Raid system on multiple workstations, and communicate reconfiguration and adaptation decisions to the servers.
- We have designed *Push*, a system with which user programs can safely and simply specify algorithms to be run within the kernel. This will allow us to evaluate the performance of protocols implemented as extensions of the operating system.

C. Related Projects

Raid is similar to Camelot [22], Argus [18], and R^* [17] in its support for distributed transactions. Camelot [22] and Argus [18] encapsulate each data object in a single server process with multiple lightweight threads of control. Raid and R^* [17] have a data server for each user. A single data server provides a performance advantage for transactions that access many data items, and amortizes session connection and authentication over several transactions. R^* uses high-level communication facilities with significant setup time, so having a relatively long-lived data server is important. The principal difference between these systems lies in the structure of the servers that process transactions. Most systems that choose to use multiple processes on each site are sacrificing the performance advantage of shared memory communication and cheap context switches for the benefits of address space protection and modularity. Camelot and Argus separate

[1] VAX is a registered trademark of Digital Equipment Corporation. Sun is a trademark of Sun Microsystems, Incorporated. UNIX is a registered trademark of AT&T Bell Laboratories.

data servers (guardians) into different processes, while Raid separates components of transaction processing in different processes. A new Raid design, described in Section II, combines multiple Raid servers into a single UNIX process to improve communication performance.

Camelot and Argus both use remote procedure call (RPC) mechanisms for communication. Camelot uses Mach RPC for invoking data server operations. Mach RPC provides high-level communication services, including automatically generated stub routines for converting arguments to and from network byte order. Mach RPC takes 30 ms for a simple call and return, while Internet Universal Datagram Protocol (UDP) roundtrips are under 10 ms. Since RPC is so expensive, UDP is used for the two-phase commit protocol. Argus handler calls are built on top of their own low-level datagram service, which can send a roundtrip datagram in 3 ms. By contrast, *R** uses a high-level error-free virtual circuit facility. Messages using this facility take 60 ms. *R** employs datagrams for recovery, but these are not low-level datagrams used for performance reasons, but special datagrams that automatically invoke a remote process to handle recovery. Raid relies on datagrams for communication. The Raid communications package provides high-level services that take almost 15 ms for a roundtrip. The package is layered on UDP which takes about half that long.

Other systems that have been developed over the years include SDD-1 [21], Encompass [13], Locus/Genesis [20], [26], Eden [15], and Sirius-Delta [3]. They have been discussed in more depth in [2].

II. Raid Architecture

We designed the software that implements transaction processing in separate server processes to provide for modularity and reconfiguration. The servers communicate among themselves using the Raid communication package. The current design provides for two versions of the Raid system. The first version runs with each server in an asynchronous process communicating via messages (see Fig. 3). The second version combines the servers that do not need to be asynchronous into a single process, thus reducing the communication cost (see Fig. 4). Transmitting a message between two servers takes tens of milliseconds in the multiple process model, but only tens of microseconds in the single process model.

The second version of Raid loads all servers which can be run synchronously into one program. When the program is started, a command-line option can be used to determine which of these servers will actually be active in the current process. (This option is useful for testing new versions of the servers. One process can be used for all the unchanged servers, while separate processes can be created for the servers being tested. This isolates the new servers and simplifies debugging.) The actual program differs from the first version in two areas: the structure of the main loop, and the implementation of the communication package. In the first version, each server has a main program consisting of a loop that receives messages and calls the appropriate subroutine to handle that message. In the second version, there is a single loop per process which receives messages for all the servers in that process. The process then uses the message header to determine which server the message was intended for and calls that server's message dispatching routine. The server then proceeds just as if it had received the message in the first version. When messages are sent, the communication package first checks whether the message is bound for a server within the same process or not. If the message is interprocess, it is sent using UDP as usual. If the message is intraprocess, it is put on a list of internal messages. The message-receive routine then checks this list for internal messages before it checks for external ones. Note that this implementation makes servers within the same process run synchronously.

Other implementations of Raid are also possible. One easy variation on the second version would be to add a priority ordering by message types to the communications package. This could give messages that might lead to a blocking state a low priority, for example. Another alternative would be to implement lightweight processes for each of the servers loaded together. This would allow asynchronous processing of the servers but retain the speed advantages of communication through shared memory rather than via UDP datagrams. We are actively investigating the possibilities of implementing these alternatives.

A. Site Organization

Fig. 1 depicts the organization of a Raid virtual site. The site is virtual since its servers can reside on one or more hosts (machines) and since the site is not tied to any particular host on the network. Each site implements facilities for query parsing and execution as a transaction, access management with stable storage, concurrency control, replicated copy management, site failure and network partitioning management, naming, etc. The following describes the role of each of the Raid servers in the system.

The communications package design supports arbitrary grouping of Raid servers into physical processes. During testing, each of the servers is a separate process to isolate errors. When performance measurements are being taken, the following two servers can be linked into a single process for each user:

User Interface (UI) is a front-end invoked by a user to process relational calculus queries specified in a subset of Quel [24].

Action Driver (AD) accepts a parsed query in the form of a procedural intermediate language from its UI and executes the transaction, reading data from the local copy of the database. It formats the query as a transaction (read and write actions).

The remaining servers can be linked together to form the transaction management process during performance experiments:

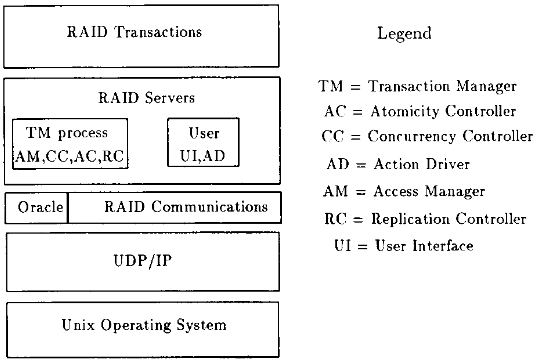

Fig. 1. The organization of a Raid site.

Access Manager (AM) provides write access to the local database, and works with AC and AD to ensure that updates are posted atomically to stable storage.

Atomicity Controller (AC) manages two commit phases of transaction processing to ensure that a transaction commits or aborts globally. When AC receives a transaction, it set a commit-lock on each data item that was accessed by the transaction. The commit-locks provide a critical section on the data items during two-phase commit. These locks are short-lived, since they are only needed during commitment, rather than during transaction processing.

Replication Controller (RC) is the server that maintains consistency of the replicated copies of the database in the event of site failures. A Read-One-Write-All (ROWAA) protocol [4] used in Raid allows transaction processing as long as a single copy is available. If a transaction on an operational site knows that a particular site is down, the transaction does not attempt to read a copy from or send an update to the failed site.[2]

Concurrency Controller (CC) checks whether a transaction history is serializable. After a transaction finishes executing, its history information is passed to all sites in the system, each of which validates whether the history is serializable with respect to previously positively validated transactions on that site. The CC has the facility to perform the validation with one of many different implementations. Currently implementations of simple locking, read/write locking, timestamping, and even serial execution are available and the choice is made by specifying an option during initialization.

B. Communications

This section describes the services provided by the communications package, including the Raid name space, the oracle (name-server), and the available communication services. Raid communications facilities are implemented on top of UDP/IP, the Internet Universal Datagram Protocol.

The Name Space: We can uniquely identify each server with the tuple ⟨instance number, virtual site number, server type, server instance⟩. To send a message to a server, UDP needs a (machine name, port number) pair. The oracle maps between server 4-tuples and UDP addresses. The communications software at each server automatically caches the address of servers with which it must communicate. Thus, the oracle is only used at system start-up, when a server is moved, and during failure and recovery.

The Oracle: An oracle is a server process listening on a well-known port for requests from other servers. The two major functions it provides are *lookup* and *registration.* A server can use the *lookup* service to determine the location of another server.

A server performs the **RegisterSelf**() call to permit other servers to locate it. RegisterSelf takes a single argument, called a *notifier set.* The notifier set is a list of regular expressions describing the addresses of servers with which the new server must communicate. Whenever a server changes status (e.g., moves, fails, or recovers) the oracle sends a notifier message to all other servers that have specified the changing server in their notifier set.

The performance of the Oracle only affects the start up and reconfiguration delays of Raid. The **RegisterSelf**(), **FindPartner**(), and **FindAll()** functions each require just a few packet roundtrips. **FindOracle()** is an order of magnitude more expensive, since it must check for the oracle on all possible hosts on the network.

Communications Facilities: The servers communicate with each other using high-level operations such as **SendAC()**. Fig. 2 shows the layering of the communications package. The fragment size is an important parameter to the LDG (Long Datagram) protocol. Normally we use fragments of 8000 bytes, which is the largest possible on our Suns. Since IP gateways usually fragment messages into 512 byte packets we also have a version of LDG with 512 byte fragments. This allows us to compare kernel-level fragmentation in IP with user-level fragmentation in LDG.

Table I compares UDP, LDG, and Raid roundtrip communication times for datagrams of various lengths. The numbers given for the Raid layer are based on LDG-8000. The current implementation copies the buffer several times while building the header. A new implementation of the Raid layer is expected to perform almost as well as LDG, because of changes that completely avoid buffer copying. Similar changes to LDG led to the current version in which LDG only requires a small constant amount of additional time over UDP.

C. Transaction Processing

Transaction processing in Raid is separated into one execution phase and two commit phases. In the execution phase the transaction executes on the site to which it was submitted, using only the local copy of the database. During this phase no concurrency control is done, and no messages are exchanged. The transaction maintains timestamps for its reads, and writes to a copy of the data in volatile memory. During the first commit phase, the executing site communicates with other sites to determine

[2] A site is *up* (operational) if *all* of its servers are operational.

High-level Raid communications (e.g., `SendAC()`)
High-level oracle interface (e.g., `RegisterSelf()`)
Low-level oracle interface (e.g., `FindOracle()`)
Low-level Raid datagrams (e.g., `SendPacket()`)
LDG (e.g., `sendto_ldg()`)
UDP (e.g., `sendto()`)

Fig. 2. Layers of the communication package.

TABLE I
COMMUNICATION TIME BY PACKET LENGTH (IN MILLISECONDS)

Bytes:	64	512	2048	8192	32768	500000
UDP	7.2	10.6	16.5	48.8	-	-
LDG-512	10.2	17.2	41.9	147.4	550.0	8,630
LDG-8000	9.6	12.8	19.2	65.1	224.8	3,200
Raid	11.9	20.1	46.7	153.3	-	-

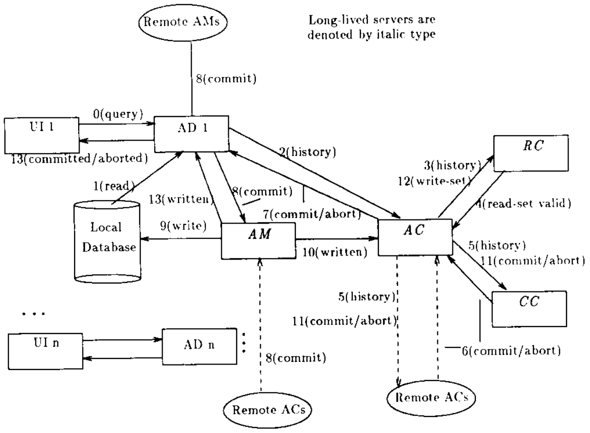

Fig. 3. Transaction processing on a Raid site (first version).

global commitment. The entire read/write set of the transaction is distributed in a single round of messages. The sites use the read/write set to determine whether the transaction should be committed or aborted. Phase 2 of commitment communicates the commit decision to all of the sites.

In order that the AC can manage multiple transactions simultaneously it is implemented in a *multithreaded* manner. A multithreaded server maintains a queue of the requests for which it is waiting for replies. Whenever the server receives a reply message, it locates the request in the queue, and updates the state information in the queue element. Whenever a server receives a request message, it immediately processes it and returns a reply.

D. Transaction Execution Flow

Fig. 3 depicts the relationships between the Raid servers during transaction processing. Transactions are processed in this manner during integration testing of Raid modules. The measurements shown in Section III-A are based on this implementation. The numeric labels in the figure refer to the phases in the life of a transaction from begin to commit as follows:

0) UI accepts a relational query from the user, parses it into a procedural intermediate language, and passes it on to AD.

1) AD assigns a globally unique ID to the transaction, and processes the transaction. Reads are performed using the local copy of the database, and writes are recorded in a log/differential file. When using locking concurrency control, reads from the AD must go through the AM.

2) AD forms a commit request and sends it to the local AC. This request contains the transaction ID, a list of identifiers of items (relational tuples) read, along with the time at which the read occurred, and the list of identifiers of items written. No timestamps are available for the writes since they have not yet taken place.

3) AC sends transaction history to RC for read-set validation if AC considers this site to still be recovering (i.e., fail-locks are still set for copies on this site). RC checks for a fail-lock on each data item in transaction's read-set. Copier transactions are generated for any out-of-date items that are found in the read-set. The implementation of the copier transactions is underway.

4) RC responds to AC with indication of read-set validity after completion of necessary copier transactions.

5) If read-set is valid (no fail-locks), AC acquires special *commit-locks* for the items accessed by the transaction. If some commit-locks are already set, it may choose to wait for them to be released, in which case it uses a method for avoiding or breaking deadlocks. AC then sends the transaction history to CC and remote AC's. If the read-set is invalid, the AC aborts the transaction.

6) CC and remote AC's reply to AC with a commit/abort decision for the transaction.

7) Once all votes are recorded from the local CC and the remote AC's, AC informs AD of the commit/abort decision.

8) AD sends the log/differential file to all AM's and tells them to commit the transaction, if the transaction was deemed globally serializable by AC.

9) AM writes all data of the committed transaction to the database.

10) AM informs AC that the transaction's data were successfully written.

11) AC releases the commit-locks belonging to the transaction, and informs the local CC and all other AC's. The remote AC's release their commit-locks and inform their CC's. The CC's move the transaction to their commit lists.

12) AC sends the transaction write-set to RC. RC clears fail-locks for items in the write-set. Fail-locks are set for any sites that are perceived to be down.

13) AM tells AD that write was successful. AD informs user that the transaction has committed.

Fig. 4 shows the communication paths between the servers in the second version, as discussed in Section II,

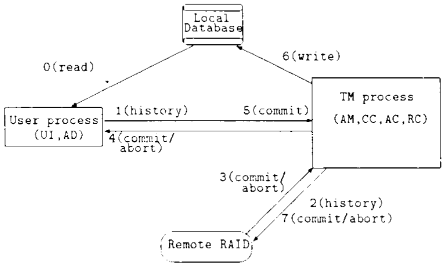

Fig. 4. Transaction processing on a Raid site (second version).

that will be used for efficient processing (still in test phase). The numeric labels refer to the phases in the life of a transaction from begin to commit as follows:

0) UI accepts a transaction from the user, and parses it as in the first version of Raid. AD interprets the parsed transaction, using the local database for reads, and recording updates in a log/differential file.

1) AD forms a commit request and sends it to the TM.

2) The TM transmits the commit request to remote TM's and invokes the local concurrency controller.

3) The remote TM's reply.

4) The local TM makes a commit/abort decision and returns it to the AD.

5) The AD acknowledges the commit/abort decision.

6) The TM updates the database.

7) The TM informs the remote TM's that the transaction is committed.

III. Experiments and Measurements in Raid

Raid provides a logging facility that automatically timestamps requested events, such as messages between servers. The logging facility is inexpensive and analysis is done offline to minimize the effects of measurement on performance.

A. *Performance Measurements in Raid*

This section describes measurements of the performance of the transaction processing protocols in the first version of Raid. The following series of performance measurements were done on Sun 3/50's (approximately 1 MIPS machines) connected by a 10 megabit/second Ethernet[3]. The database for the experiment is 100 tuples from a truncated version of the thousand relation used in the benchmark described in [12]. For example, the query **database thousand, get thousand : thousand.ten = 8;** can be used to select approximately 10 percent of the tuples.

1) Elapsed Time for Transaction Processing: Table II shows the time taken by transaction processing for several different database queries on Raid systems with varying numbers of sites. The times do not include the cost of interpreting the database query or the cost of translating the query to a transaction.

TABLE II
Execution Time for Transactions (in Seconds)

transaction	1 site	2 sites	3 sites	4 sites
select one tuple	0.3	0.3	0.4	0.4
insert twenty tuples	0.6	0.6	0.8	0.8
update one tuple	0.4	0.4	0.4	0.4

The fact that the processing time is fairly constant as the number of sites increase is due to the use of built-in multicast in the Raid layer of the communications package [7], [8]. This lower level multicast only has to format the packet once regardless of the number of sites. Hence, the execution occurs in parallel on each site. Our estimate is that this time will remain constant up to around ten sites if we continue to use UDP as our transport mechanism. We have developed two kernel-level multicast mechanisms [7] that will help maintain this property for even larger numbers of sites.

2) Server CPU Time: A significant fraction of the CPU time is spent processing messages. For instance, the CC spends 40 milliseconds of CPU time processing a simple transaction, about 10 milliseconds of which are spent in a single message roundtrip. The total CPU time for all of the servers is a small fraction of the elapsed time for each transaction. This suggests that multiple queries executing at the same time would be able to overlap significantly. CPU times for most of the servers are constant as the number of sites increases, but the AC does some additional processing for each new site. Table III shows the CPU time taken by the AC for various numbers of sites. The times show a slight tendency to increase with the number of sites, but the variance in the measurements is too large to permit stronger statements.

One main conclusion that we draw from this work is that the I/O and communication times dominate the processing time. Using the best algorithms for concurrency control or manipulating data structures in the best possible manner does not show up in the bottom line.

B. *Replicated Copy Control Experiments*

This set of experiments examines the effects of site failures and recoveries on the consistency of replicated copies by measuring how fast consistency can be restored and the overhead associated with replicated copy control algorithms. Experiments were conducted using database size, transaction length, and number of sites as independent variables [9], [5]. Here we present our results briefly.

The experiments were run on mini-Raid [9], which uses the ROWAA protocol [4] and a strategy that includes concepts of session vectors and fail-locks to maintain the consistency of replicated data during site failure and recovery. Each entry of the session vector maintained at each site represents the number of times a particular site has recovered. Fail-locks maintained at an operational site are

[3] Ethernet is a trademark of Xerox Corporation.

TABLE III
CPU TIME USED BY ATOMICITY CONTROLLER (AC) (IN SECONDS)

	1 site		2 sites		3 sites		4 sites	
transaction	user	sys	user	sys	user	sys	user	sys
select one tuples	0.04	0.14	0.06	0.14	0.04	0.10	0.08	0.24
insert twenty tuples	0.20	0.16	0.08	0.14	0.10	0.12	0.08	0.10
update one tuple	0.04	0.10	0.06	0.12	0.06	0.16	0.06	0.16

used to identify out-of-date items on a recovering site. We found that the overhead for a control transaction that announces failures or recovery by manipulating the session vectors is comparable to the cost of a small database transaction. The copier transactions that refresh out-of-date copies (or clear fail-locks) add 50 percent overhead to the user transactions. To reduce this overhead one design principle is to divide the recovery process into two steps. Before beginning the first step, a recovering site computes the percentage of the frequently referenced copies that are fail-locked. If this percentage is greater than some fixed value, the recovering site enters step one. Otherwise the recovering site omits the first step and enters step two. In the first step copies are refreshed by normal transaction processing or by copier transactions induced when a user transaction attempts to read a fail-locked item. Once the percentage of fail-locked copies drops below the fixed value, the site starts step two. In the second step the recovering site begins to issue copier transactions in a "batch" mode for all fail-locked items.

In another experiment, detailed in [5], we measured how setting the threshold level in a partially replicated system can affect data availability. The threshold is specified at system configuration time. It refers to the minimum number of copies of each object to be maintained in the system. Availability is measured in terms of the number of transactions aborted because of site failure and the resulting unavailability of data. The Mini-Raid system, extended for partial replication, was used for this experiment. Systems with different threshold levels were used. In each system, half the sites were failed and sets of transactions were processed as more failures and recoveries occurred. In a sample experiment, a 12 site system with a degree of replication of 3 was started up. The maximum transaction size used was set to 5, and the number of frequently referenced items in the database was set to 100. The experiment was carried out for three different threshold levels of 1, 2, and 3. This experiment has shown that increasing the threshold level in a partially replicated system can improve availability. There is a knee in the threshold versus availability curve at a threshold of three. Increasing the threshold above this value yields a substantially smaller improvement in availability.

There are several design principles that we learned from these experiments. When deciding on the degree of replication for a distributed system, two of the main concerns are transaction throughput and data availability. When comparing partially and fully replicated databases, it is true that partial replication requires remote access of nonlocal data, but this cost is low compared to the costs saved in the write operation. Namely, these savings are due to the reduction in message sizes, disk accesses, and the number of copier transactions. The other fear with having a low degree of replication is the nonavailability of data. Our experiments show [5] that maintaining the threshold automatically increases the availability significantly. In a 12 site system with a degree of replication of 3 and a threshold of 3, 9 site failures caused only 5 percent of the transactions to be aborted. The top graph in Fig. 5 shows that many fail-locks get set in a fully replicated system. We note that if the number of failures is increased, fail-locks will be set on more sites causing substantial delays. Therefore, when designing a distributed database system, partial replication with automatic copy generation should be seriously considered.

C. Communication Experiments

The clock granularity on our Sun 3/50's is 20 milliseconds and the communications services in our experiments take less than 10 milliseconds. Most of our measurements are performed using a *ping* protocol. A single ping roundtrip consists of a message and a reply. We measured the time required for several thousand round-trips, and averaged to determine the time for a single roundtrip. This work was reported in full in [7].

1) Ethernet Measurements: The experiments measure the roundtrip times for small datagrams, such as those used for distributed commitment. The experiments were done using UDP and SE (Simple Ethernet), a suite of low-level network services that we developed [8]. Table IV summarizes the costs of various components of datagram communication. The "Time" column gives the delay introduced by each component.

The lesson from this study is that decreasing the device layering time needed to perform a kernel service is an effective way to improve datagram speed. An alternative to directly decreasing the layering cost is to decrease the number of times the cost is incurred. For instance, multicasting can transmit many datagrams with a single system call.

2) Local IPC Measurements: One problem with using UNIX for building database systems is the poor performance of the Interprocess Communication (IPC) mechanisms [27]. We measured the performance of four IPC extensions to UNIX. The results of our measurements are shown in Table V. Message passing using queues incurs $\frac{1}{3}$ to $\frac{1}{2}$ the delay of UDP, depending on the size of the message. Shared memory with semaphores took substantially more time than message passing. This result was especially surprising since the shared memory approach only copied the data into the shared segment and not out of it, while the message passing implementation copies data both ways. Omitting the copy from the 10 byte shared message experiment did not change the result, indicating that almost all of the elapsed time is due to the semaphore

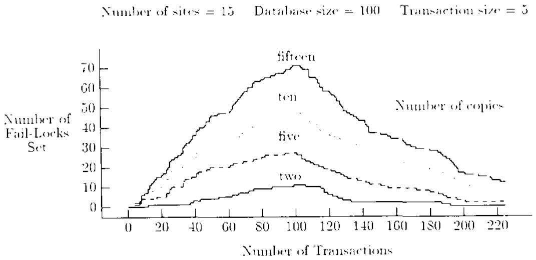

Fig. 5. The impact of replication on data availability.

TABLE IV
MEASUREMENTS OF THE SE SERVICE COMPONENTS

Protocol Component	Time (in ms)
user-level code	0.08
system call	0.33
device layering	0.61
kernel transmit	1.36
mbufs on read	0.45
context switch	0.40
kernel-to-user copy	0.30

TABLE V
LOCAL COMMUNICATION COSTS (IN MILLISECONDS)

	MESSAGE SIZE	
METHOD	10 Bytes	1000 Bytes
2 Q Message Passing	2.0	2.9
1 Q Message Passing	2.0	2.9
Named Pipes	2.3	3.9
Shared Memory	5.1	5.5
UDP Communication	4.3	9.6

operations. The high cost of semaphores is probably due to the complicated semaphore semantics. In particular, simple semaphores can be implemented more cheaply as message queues with one byte messages!

3) Multicast Measurements: We sent messages to multiple destinations according to three different approaches: kernel-level simulated multicast, physical multicast, and user-level simulated multicast. The results of the experiments are summarized in Table VI.

The results indicate approximately a 38 percent improvement in performance of the kernel-level simulated multicast facility when compared to the user-level simulated multicast. In both cases the time increases linearly with the number of destinations. However, the extra cost per destination for user-level multicast is 1.2 ms instead of the 0.75 ms for the kernel-level case. Both the user-level and kernel-level multicast simulations use our SE protocol [8]. SE is already a stripped down protocol, performing roundtrips twice as fast as UDP. Performing the experiment with UDP would add an additional 1.5 ms per destination for the user-level multicast, while adding only a constant 1.5 ms to the kernel-level multicast. Sending to ten destinations, the user-level multicast would take 2.7 * 10 = 27.0 ms, while the kernel-level multicast would take 2.7 + 0.75 * 9 = 9.45 ms, for a savings of over 50 percent. For systems like Raid, we plan to study how these new ideas of implementing multicasting affect performance.

TABLE VI
MULTICAST TIMING FOR VARYING NUMBER OF DESTINATIONS

Number	Time (ms)		
of	kernel	hardware	user
destinations	level		level
1	1.2	1.2	1.2
5	4.2	1.2	5.9
10	8.0	1.2	11.7
15	11.7	1.2	17.5
20	15.4	1.2	23.4

D. Concurrent Checkpointing Experiments

To allow continuity of transaction processing in the Raid system, we must deal with the failure and restart of individual servers. To recover from failures, a global consistent state must be checkpointed distributively. In addition, the restoration to a previous global state must be synchronized. We experiment with our algorithm [16] that allows concurrent and robust checkpointing and recovery in a distributed system. The experiment measures the elapsed time and cpu usage of a process during the execution of the algorithm. Elapsed time denotes the delay in terms of the time that a process spends during the synchronization of checkpoint operations or rollback operations with other processes. Details of these experiments are given in [6].

Checkpoint delay is the time to write the image of a process into the disk, while rollback delay is the time to read the image of a process from the disk. We have examined about 900 object files on a local UNIX system, some of which are system files, while others are user files. These cover over 90 percent of all the object files in the UNIX system. The size of these object files (excluding their text segments) ranges from 4K bytes to 48K bytes. The checkpoint and rollback were measured to take time ranging from 89 ms to 496 ms. The size of a synchronization message is 22 bytes.

1) Measurements of Elapsed and CPU Times: In our experiments that were performed on mini-Raid, several server processes communicate through message queues in Sun UNIX. A coordinator initiates a synchronization instance, and sends request messages to some participants. Each experiment involves the following steps:

1) Execute normal processes which send normal messages to one another.

2) Invoke a checkpoint starter or a rollback starter which sends a special message to designated processes. A process that receives this message initiates a checkpoint instance or a rollback instance, respectively.

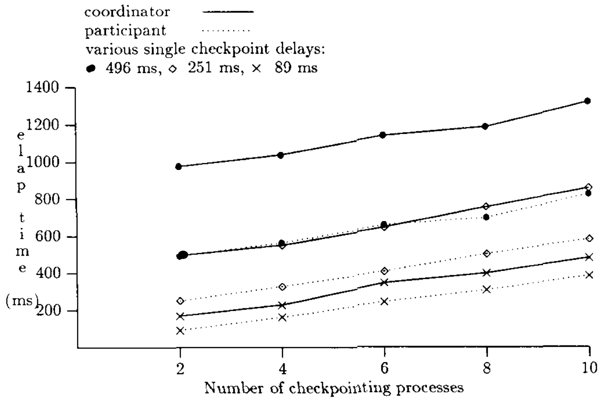

Fig. 6. Elapsed time for checkpoint processes.

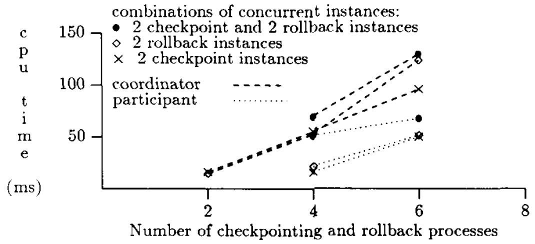

Fig. 7. CPU usage in concurrent checkpointing and rollback.

3) Run a special command that stops the normal processes.

In the experiments we measure the performance of the coordinator and participants separately during the execution of the algorithm. A synchronization instance has two to ten processes. Fig. 6 shows elapsed time for checkpoint instances with respect to three different single checkpoint delays: 89 ms, 251 ms, 496 ms.

In Fig. 7 we show CPU usage in the following three cases: a) two checkpoint instances interfere with each other, b) two rollback instances interfere with each other, and c) two checkpoint instances and two rollback instances interfere with each other.

Elapsed time of the coordinator gives us an idea about the interval between checkpoint instances. If the coordinator spends t (elapsed) time to checkpoint, then the coordinator should wait for at least t time before initiating a second checkpoint instance. In the worst case, the message overhead of the coordinator can increase quadratically as the number of participants increases. The measurements lead to the conclusion that concurrent execution does not reduce the message overhead or cpu usage. However, concurrent execution can reduce the synchronization delay by about 100 percent if all the instances are initiated at about the same time and have the same participating processes.

IV. Future Work and Conclusions

Our plan is not limited to just building a distributed database system and making claims on its implementation and performance. Instead we plan to investigate a variety of algorithms, evaluate their performance in our laboratory, identify useful observations and design principles, and finally to implement them in the Raid system. The ultimate hope is that various experiments can be designed, performed, and analyzed to give insight to other builders of such systems. Currently we are limited by the lack of good benchmarks for transaction processing, models for failures, measurement tools, and by the use of UNIX for our environment.

Our measurements and experiments that have been completed so far will provide input parameters for future experiments. In the process, we have developed excellent facilities and software for doing our future experimental work. We are planning the following experiments.

1) We plan to study the implementation and performance of transaction processing protocols in the operating system kernel and measure the performance enhancements. We plan to implement the Push system described briefly in Section IV-A.

2) We plan to evaluate the dynamic quorum algorithms for replicated copies. We will experiment with network partitioning and multiple site failures.

3) We plan to perform measurements with the new implementation of Raid where several servers reside within the same process.

4) We plan to identify the cost of dynamic reconfiguration and adaptation. What are the overheads associated with synchronizing multiple accesses to the state information? Is throughput reduced during adaptation? Under what conditions is adaptability beneficial?

A. Kernel-Level Execution of User-Defined Protocols

The need for the implementation of transaction processing protocols in the operating system kernel has been discussed in [25]. We have designed software called PUSH [7] to provide an experimental platform for measuring the benefits of such kernel support. We briefly describe the mechanisms in terms of network services, although it is applicable to other operating systems services, such as process management, memory management, and file system buffer management.

Our approach is to dynamically load special procedures into the kernel to process complete communication protocols with a single system call. These procedures will be written in a simple language that can be interpreted by the kernel in an efficient but safe manner. The approach is similar to the packet filter described in [19], in which user specified code can be dynamically loaded into the kernel to demultiplex packets for user-level implementations of network protocols. Our routines would be able to collect multiple messages, generate response packets, and only return to the user once for a complex interaction. For instance, a multiphase commit protocol could be written in this language that would send and receive two messages for every site in the system with a single system call.

A KUP (Kernel-level execution of User-level Protocols) language is one in which a user can specify complex service requests that will be executed entirely within the kernel. KUP programs are invoked with special system calls. They can issue read and write requests using existing network services such as UDP, IP, or SE [8]. A KUP language must satisfy the conflicting goals of providing efficiency, maintaining address space protection within the kernel, and preventing one process from excluding others from the CPU.

We have designed a particular KUP language called Push. Push is a simple stack-based language which can be interpreted efficiently within the kernel. In [7] we describe the language, and give two example Push programs, one for kernel-level multicast and one for a multiple RPC call. In [7] we estimate the performance improvements expected from Push based on the communications software measurements in [8]. We find that MultiRPC calls using UDP could be performed at least 25 percent faster using Push. We also compared the cost of the Push implementation of simulated multicast with the C language implementation described in [7]. Another potential use of the Push language is for applications that require guaranteed response times. For instance, many fault-tolerance protocols require the detection of failed hosts or networks within a bounded time. Also, real-time applications require the responses arrive before a specified time.

In the near future, we will have answers to the following questions. What is the impact of a KUP implementation on kernel size and performance? What other kernel support is needed for distributed processing? We want to implement Push and use it to experiment with other kernel-level protocols.

B. Conclusions

Raid has been successful in developing a software infrastructure that promotes experimentation. Central to this capability is the server philosophy, which clearly defines the interfaces to the functional components of the system. New implementations need only match this interface to run correctly as Raid servers. Current performance problems are caused both by the additional operating system overhead of managing multiple processes, and by the expensive communication primitives available. To correct these problems we have modified the design to provide for a high-performance single-process version of Raid.

We made an early decision that the communications system would be datagram based. This works well for small control messages, but for messages containing large amounts of data it uses excessive resources. For instance, the log or differential file that is transmitted to the Access Manager to be merged with the database must be placed in a single large buffer, fragmented by LDG, transmitted in pieces by UDP/IP, placed into a single large buffer, and finally passed to the AM. Our future implementation will transmit the log as a stream or as a series of datagrams, and the AM will process and discard each fragment as it receives it.

The need for concurrent execution in several server types has been a problem. The AD must be able to process several user transactions concurrently. We provide for this by creating a different AD for each user. This approach has the disadvantages that it creates many new processes between which UNIX must context-switch, that multiple copies of the code for executing transactions must be loaded into memory, and that database caches can only be kept on a per-user basis. The AC supervises distributed commitment for multiple transactions simultaneously. Currently it maintains the state for each transaction internally, looking up the corresponding transaction for each message it receives. Since this approach does not provide for any form of preemptive scheduling, it is essential that the AC process each message quickly, so that other incoming packets are not discarded. The AC implementation would be simplified if the operating system provided preemptive scheduling within the AC address space. In addition, the CC could be modified to process multiple transactions simultaneously. Each of these servers would benefit from operating system support for multiple lightweight processes within the same address space. We are planning an empirical investigation of the implementation and performance advantages of lightweight process support.

Currently our processing times are around 400 milliseconds for one to seven reads and 500 milliseconds for writes. Our times are slightly higher than CAMELOT times [23], since we deal with an action that reads or writes in tuple in a database relation and not just a single value. In addition, we use a separate server for each function on a site for the measurements, so our timings include the overhead of communications among the servers. For more discussion see [2].

Acknowledgment

The following students have contributed to the Raid project at the University of Pittsburgh and at Purdue University: T. Chen, J. Dilley, A. Helal, F. Lamaa, H. Lee, P. Leu, S. Leung, E. Mafla, T. Mueller, P. Ng, P. Noll, A. Royappa, D. Sabo, and B. Sauder.

E. Mafla, T. Mueller, and B. Sauder performed the communications experiments. P. Noll and F. Lamaa did the replicated copy experiments. P. Leu performed the checkpointing experiments.

We would like to thank Dr. J. Gray of Tandem Corporation for suggestions that led to the new design in which multiple servers share a single operating system process. We would like to thank Dr. L. Lilien of AT&T Bell Laboratories for his help in preparing the first draft of this paper in the Summer of 1987.

We would also like to thank the anonymous referees for their suggestions that led to a more concise presentation.

References

[1] Anon *et al.* "A measure of transaction processing power," *Datamation*, vol. 31, no. 7, pp. 112-118, Apr. 1985.

[2] B. Bhargava, "Building distributed database systems," *Int. J. Inform. Sci.*, to be published.

[3] B. Bhargava, Ed., *Concurrency and Reliability in Distributed Systems*. New York: Van Nostrand Reinhold, 1987.

[4] B. Bhargava, "Transaction processing and consistency control of replicated copies during failures," *J. Management Inform. Syst.*, vol. 4, no. 2, pp. 93-112, Oct. 1987.

[5] B. Bhargava, F. Lamaa, P. Leu, and J. Riedl, "Three experiments in reliable transaction processing in RAID," Purdue Univ., Tech. Rep. CSD-TR-782, June 1988.

[6] B. Bhargava, P. Leu, and S. Lian, "Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms," Dep. Comput. Sci., Purdue Univ., West Lafayette, IN, Tech. Rep. CSD-TR-790, July 1988.

[7] B. Bhargava, E. Mafla, J. Riedl, and B. Sauder, "Implementation and measurements of an efficient communication facility for distributed database systems," in *Proc. 5th IEEE Data Engineering Conf.*, Los Angeles, CA, Feb. 1989.

[8] B. Bhargava, T. Mueller, and J. Riedl, "Experimental analysis of layered Ethernet software," in *Proc. ACM-IEEE Comput. Soc. 1987 Fall Joint Computer Conf.*, Dallas, TX, Oct. 1987, pp. 559-568.

[9] B. Bhargava, P. Noll, and D. Sabo, "An experimental analysis of replicated copy control during site failure and recovery," in *Proc. 4th IEEE Data Engineering Conf.*, Los Angeles, CA, Feb. 1988, pp. 82-91.

[10] B. Bhargava and J. Riedl, "A model for adaptable systems for transaction processing," in *Proc. 4th IEEE Data Engineering Conf.*, Los Angeles, CA, Feb. 1988.

[11] B. Bhargava, J. Riedl, and D. Weber, "An expert system to control an adaptable distributed database system," Purdue Univ., Tech. Rep. CSD-TR-693, May 1987.

[12] D. Bitton, D. DeWitt, and C. Turbyfil, "Benchmarking database systems: A systematic approach," in *Proc. VLDB Conf.*, Oct. 1983.

[13] A. Borr, "Transaction monitoring in Encompass," in *Proc. 7th Conf. VLDB*, Sept. 1981.

[14] A. Helal, J. Srinivasan, and B. Bhargava, "SETH: A quorum-based replicated database system for experimentation with failures," in *Proc. 5th IEEE Data Engineering Conf.*, Los Angeles, CA, Feb. 1989.

[15] W. H. Jessop *et al.*, "The Eden transaction-based file system," in *Proc. Second IEEE Symp. Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1982.

[16] P. Leu and B. Bhargava, "Concurrent robust checkpointing and recovery in distributed systems," in *Proc. 4th IEEE Data Engineering Conf.*, Los Angeles, CA, Feb. 1988, pp. 154-163.

[17] B. G. Lindsay, L. M. Haas, C. Mohan, P. F. Wilms, and R. A. Yost, "Computation and communication in R*: A distributed database manager," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, Feb. 1984.

[18] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, "Implementation of Argus," in *Proc. 11th ACM Symp. Operating Systems Principles*, Nov. 1987.

[19] J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The packet filter: An efficient mechanism for user-level network code," in *Proc. 11th ACM Symp. Operating Systems Principles*, Nov. 1987.

[20] G. J. Popek and B. J. Walker, *The LOCUS Distributed System Architecture*. Cambridge, MA: MIT Press, 1985.

[21] J. Rothnie *et al.*, "Introduction to a system for distributed databases (SDD-1)," *ACM Trans. Database Syst.*, vol. 5, pp. 1-17, Mar. 1980.

[22] A. Z. Spector *et al.*, "The Camelot project," *Database Eng.*, vol. 9, no. 4, Dec. 1986.

[23] A. Z. Spector, D. Thompson, R. F. Pausch, J. L. Eppinger, D. Duchamp, R. Draves, D. S. Daniels, and J. J. Block, "CAMELOT: A distributed transaction facility for MACH and the Internet—An interim report," Dep. Comput. Sci., Carnegie-Mellon Univ., Tech. Rep. CMU-CS-87-129, June 1987.

[24] M. Stonebraker, Ed., *The INGRES Papers*. Reading, MA: Addison-Wesley, 1986.

[25] M. Stonebraker, "Operating system support for database management," *Commun. ACM*, vol. 24, no. 7, pp. 412-418, July 1981.

[26] J. Thomas, W. Page, M. J. Weinstein, and G. J. Popek, "Genesis: A distributed database operating system," in *Proc. ACM-SIGMOD 1985 Int. Conf. Management of Data*, May 1985, pp. 374-387.

[27] P. J. Weinberger, "Making UNIX operating systems safe for database," *Bell Syst. Tech. J.*, vol. 61, no. 9, Nov. 1982.

Bharat Bhargava (M'87-SM'87), for a photograph and biography, see this issue, p. 662.

John Riedl received the B.S. degree in mathematics from the University of Notre Dame, Notre Dame, IN, in 1983, and the M.S. degree in computer science from Purdue University, West Lafayette, IN, in 1985.

He is currently working toward the Ph.D. degree in computer science at Purdue. His research interests include distributed database systems, distributed operating systems, and communications.