BUILDING DISTRIBUTED DATABASE SYSTEMS†

by

Bharat Bhargava
Computer Science Department
Purdue University
West Lafayette, Indiana 47907
(317) 494-6013

† This research is supported by NASA and AIRMICS under grant number NAG-1-676, U.S. Department of Transportation contract number DTRS-57-85-C-00099, Unisys Corporation and David Ross Fellowship.

BUILDING DISTRIBUTED DATABASE SYSTEMS†

by

Bharat Bhargava
Computer Science Department
Purdue University
West Lafayette, Indiana 47907
(317) 494-6013

ABSTRACT

Several design principles necessary to build high performance and reliable distributed database systems have evolved from conceptual research, prototype implementations, and experimentation during the eighties. This paper focuses on the important aspects of transaction processing, including: communication, concurrency, atomicity, replication, and recovery as they apply to distributed systems. Implementation of these in eleven experimental and commercial distributed systems are briefly presented. Details of a twelfth system called RAID that has been implemented by us are included to show our design and implementation strategies. The relationship between database and operating systems is discussed along with the desirable features in communication software for reliable processing. This material has been presented to demonstrate the practicality of certain successful design and implementation choices so as to benefit those who have the responsibility for making distributed systems work.

† This research is supported by NASA and AIRMICS under grant number NAG-1-676, U.S. Department of Transportation contract number DTRS-57-85-C-00099, Unisys Corporation and David Ross Fellowships.

1. INTRODUCTION

Distributed database systems are needed for applications where data and its access are inherently distributed and to increase availability during failures. Prime examples of applications are systems for international air-line reservations, financial institutions, and automated manufacturing. Database system designers also choose to replicate and distribute the data to increase availability during failures. However, distribution gives rise to additional vulnerability to failures and increases the complexity of implementations.

There have been several research efforts that have successfully built prototype distributed database systems namely SDD-1 [44], Distributed Ingres [49], System R* [35], Encompass [17], Sirius-Delta [6,28], and Raid [13]. Many research ideas on the subject of concurrency control, reliability, commitment, and transaction processing have evolved through these efforts. Several attempts in distributed operating/programming systems Locus [40], Clouds [22], Mach/Camelot [42,47], Argus [36], Eden [32], Isis [15], have also contributed towards the implementations of the distributed database systems. These have also added to the understanding of the relationship between database and operating systems. In addition, we have gained experience with object oriented systems. Some of these systems are commercially available while other are in academic settings.

There are many research issues and concepts that must be considered in the design of a distributed database systems. They include naming, communications, centralized vs. decentralized control, concurrency, reliability, replication, atomicity, and concepts of transaction and nested transaction. In addition the techniques of data distribution and query optimization are needed. Fortunately a large body of knowledge exists in the literature and several books [2,6,18,39] have been written on the subject.

In this paper, we focus on understanding how these concepts have been found useful in building distributed database system and providing the support for transaction management. We will discuss the design issues as well as the implementations of several prototypes. In particular, we will give insights into the implementation of a distributed database systems called Raid that has been implemented by us. Finally based on our experience, we discuss the enhancements in operating systems that can be useful for database systems.

2. DESIGN ISSUES

We briefly outline the concepts that are necessary for implementation. They include the structure/architecture of the system, communication software, concurrency, replication management during failure, and heterogeneity. Since a major component of a distributed system is communication support, it is discussed in depth.

2.1. Structure

There are two system structures that can be distinguished: *Object* based and *Server* based. In object based structure, the processes manage objects that encapsulate data. For

example a process may manage the concurrent access to an object or provide recovery for it. A transaction is no longer responsible for ensuring these properties. The objects are a convenient mechanism for the abstraction and isolation of data and encapsulation of operations on them. The operations are activated through invocations by active processes. Sometimes the object management, remote and local invocations are provided at the level of system kernel. Applications are written using calls to appropriate objects. Examples of object based systems are Clouds, Eden, Isis, and Argus.

In the server based structure, processes act as servers for particular functions. For example a concurrency control server can preserve the serializability of all transactions being served by it. The recovery control server will make sure that all database items are written on the permanent storage in a safe manner. Examples of server based systems are SDD-1, Distributed Ingres, and Raid.

2.2. Communication Services and Primitives

Because of performance consideration, many of the distributed systems build their own special communication software. In communication subsystems, low level and high level services and primitives are provided. We outline the choices of transport service, the choices of communication primitives, and their semantics.

2.2.1. Low Level Transport Service

The transport service provides a data transfer facility between processes. There are two conceptual models. *Virtual circuits* provide reliable transfer of data packets between two processes, with some form of flow and error control, while *datagram* provide delivery of single packets from a source process to a destination process on a "best effort" basis. The choice of virtual circuit or datagram strongly influences the other design decisions.

Many distributed systems such as Locus use datagrams, basically for performance reason. Their underlying networks are fairly sound so that the probability of successful delivery of packets is very high. Moreover, in many cases the transport service is used only by special software to support remote procedure call (RPC) in which the inter-process communication is mostly in form of request/response. Flow control and error handling can be integrated with higher level service. The connection management and lower level acknowledgements used by virtual circuit introduces unnecessary overhead. However, the special software at the upper layer must be responsible for dealing with loss, duplication and misordering of requests/responses.

Some distributed systems choose virtual circuits for their transport service. As an example, the communication in R*, is built on top of the IBM SNA (Systems Network Architecture), a virtual circuits protocol. Virtual circuits not only handle the flow and error control, but also facilitate failure detection and resolution. Based on the model that a user computation is a tree of processes connected by virtual circuits, R* uses a very simple failure-resolution mechanism, which is safe unless the failure is detected during the commit procedure in which case, R* uses datagrams as well.

2.2.2. Communication Primitives

There are two basic types of communication primitives: *message passing* and *remote procedure call* (RPC). Systems such as Camelot, Distributed Ingres, and Locus use such primities as *send* and *receive*, while R* and Argus use RPC as their basic means of inter-process communication.

The use of message-based primitives has its advantage in flexibility of supporting different programming languages and applications. But it introduces a new mechanism of inter-module interaction in distributed environments, syntactically and semantically different from procedure calls used for inter-module interaction in single processor systems. It is also error-prone because the send and receive primitives are syntactically unrelated. In contrast, RPC shares not only the same syntax but also certain properties such as compile-time type checking, with conventional procedure call. However, it is difficult for RPC to have the same semantics as the conventional procedure call because of the possibility of processor and communication failure. For instance, it is nontrivial to ensure that every RPC always returns. We will discuss more on RPC semantics below.

Inter-process communication can be either *synchronous* or *asynchronous*. In fact, the same terms are used at different levels in the literature. In the message passing level, the synchronous send means that the sender is blocked until the message is received by the destination process. It not only transfers data but synchronizes the sender and receiver. The asynchronous send, or *no-wait send*, means that the sender resumes as soon as the message is out of its hand (possibly buffered by the network). It achieves a high degree of parallelism but causes problems such as congestion control. Most of the literature talks about synchronism at an even higher level, e.g. at the RPC level. That is, synchronous communication means that the requesting process blocks until it receives the response, while the asynchronous communication sends requests without waiting for responses.

2.2.3. Reliability Semantics of Communication Primitives

In the absence of processor or communication failures, the semantics of a communication primitive are well defined. An RPC, for example, can have the same semantics as a local call, except that it takes place on a remote processor. However, system designers concentrate more on their semantics in the presence of failures, called *reliability semantics*.

From the reliability point of view, the message passing primitives vary from system to system. On one extreme end, the message passing primitives built directly on unreliable datagrams provide high performance but low reliability. The upper-level software using such primitives must be responsible for detecting and handling transport errors, as discussed before. On the other end, some systems have their message passing primitives supported by communication subsystems to enforce reliability. For example, the RelNet (Reliable Network) of SDD-1 is a virtual network, hiding the unreliable real network from the upper-level software. It guarantees reliable message delivery even if the sending site and the destination site are never up simultaneously.

For remote procedure calls, the reliability semantics concentrate on the problem of duplicate requests. Because it is not desired to abort an entire transaction when a calling process has not heard from its callee within a timeout period, many systems retransmit requests. But what effect on the callee site is expected when the call eventually returns?

The *exactly-once* semantics means that if the client gets positive response from the server (i.e., the RPC returns successfully), exactly one execution has taken place at the server site. Sequential numbering is used for duplicate suppression.

Some systems choose the *at-least-once* semantics. That is, the positive response from the server (i.e. success of the RPC) implies that at least one execution has taken place at the server site. This semantics is acceptable only if the requested operations on the server site are *idempotent*, i.e., repeated executions of the same operation are equivalent to a single execution. File access operations can be tailored to satisfy that requirement. However, not all operations can have such a property. There are various subclasses of the at-least-once semantics, for example, to ensure that the response received by the client is that issued by the very last execution.

Some systems such as Argus and Clouds implement remote procedure calls as atomic actions, called the *at-most-once* semantics. That is, a successful return of a RPC implies that the operation has taken place at the server site exactly once. If the RPC fails, or is canceled by the caller, it has no effect at all. Using RPC with at-most-once semantics releases programmers from the burden of dealing with partial failures. This is a good approach to achieve reliable computation in distributed systems, but it is not necessary for all remote operations to have such an atomic property.

2.2.4. Models for Remote Service Control

A client program invokes operations on remote sites and obtain service or data. Database applications require facilities such as synchronized clocks, atomic commitment, mutual consistency among replicated data. Each of these facilities need communication support for broadcast, multicast, datagram, and virtual circuit functions. There are several models for remote service control flow. Three of them are: Procedure model, Process model, and Client/Server model. They are briefly discussed in the following.

The Procedure Model.

The basic idea of the procedure model is that a service should be provided by a simple procedure run as part of the calling process. Essentially, a service is requested by a procedure call on an appropriate library routine. The principle advantage of the procedure model is its simplicity and a well understood interface to the user of the service. The disadvantage is that the semantics of this model are not as clean as the syntax. In particular, the assumption that a service is part of a user process is often undesirable, perhaps for security or record keeping. In a distributed system the problem is even more acute. The semantics of the single site procedure call are fairly difficult to preserve across machine boundaries, since these semantics are grounded in the mechanics of the procedure call on a single machine. For this reason, most of the distributed systems that appear to be based on the procedure model are actually layered on top of one of the other methods. Unfortunately, this layering has proven very difficult to completely disguise.

In a distributed system, the implementation of the procedure model is difficult. Most designers have chosen to layer the procedure call interface on top of some other method. Unfortunately, there is not yet an agreement on the semantics of a procedure call, mostly because the desired semantics (precisely those of the single machine procedure call) are fairly difficult to implement. Furthermore, as mentioned above, the

layering often becomes visible to the user in these systems, which ruins the primary advantage of the method.

The Process Model.

Our definition of the process model is that a separate process is established for each service request. Such a process receives arguments from the parent process and possibly returns results, but is otherwise distinct.

The principal advantage of the process model is that the communication paths between the requester and the server are significantly reduced. In particular, in any machine with an address space protection mechanism, the service routine can be completely insulated from the calling routine.

The disadvantage is that this insulation can be costly. Only a very few operating systems are able to create processes in a reasonable multiple of the time in which they can execute a procedure call. In fact, a fairly convincing argument can be made that conventional process creation must take considerably longer than the corresponding procedure call. The semantics of this method are quite complex. For instance, if the parent procedure dies due to an outside signal, we have the orphan problem [51].

The Client/Server Model.

In the client/sever model each service is supplied by a separate, permanent process, called a server. Requests are communicated to the server via messages, and the server sends back information via messages. Both synchronous and asynchronous client/server systems have been developed. Modern implementations of this model often use object-oriented design principles. This means that the servers enforce abstract data typing, by only allowing correct operations on objects of each type. This model is used by Argus and Camelot.

The principal advantage of the client/server model is that it is a natural way to implement distributed systems across multiple autonomous sites, each of which wants to perform request validation. More recently the advantage of having the server and the client in separate address spaces without the high cost of process creation for each request has also been recognized.

Unfortunately, modularity in the client/server model has proven to be a problem. This is because the validation overhead for each request penalizes tasks that require several layers of request. The problem is even more severe in object-oriented systems. For example, attempt to lower this overhead, R* creates a separate server process for each user, and only checks the users' privileges one time.

In distributed or multiprocessor systems, the implementation of the client/server model is perhaps the most natural of those we have discussed. The most significant difficulties lie in constructing efficient usable servers that work correctly in the face of loss, destruction, or arbitrary delays of messages. *Stateless* servers help in getting over these difficulties. A server is stateless if its response to a client request does not depend on the history of previous requests [21].

2.3. Concurrency

Multiple users can submit transactions at different sites of the system. The concurrency control subsystem provides a serializable history for all transactions. Serializability requires that the effect on the database is the same as if each transaction completed before the next one started. In some applications a lower level of consistency which does not require serializability may be acceptable. Many systems have successfully implemented concurrency control mechanisms.

There are basically three types of algorithms for concurrency control: Locking based [30], Time-stamp based [4], Version based [43]. They have been surveyed and compared in [2,3,5,39]. Locking has been found to be more restrictive in terms of the degree of concurrency than the time-stamp based methods. However, implementors have found locking to be easier to implement. In many distributed database system, locking has been implemented using centralized control. One central site maintains a lock table and is responsible for granting and releasing global locks. A popular and easy to implement algorithm is called two-phase locking. Time-stamps have been used to provide high degree of concurrency and have been implemented in several ways [4]. Transactions can be assigned a single time-stamp as in SDD-1 or a vector of time stamps [34]. Time-stamps can also be assigned to database objects [53]. A validation (certification) procedure is required before any transaction can commit. The validation can be done at the time when transaction begins to execute, during its execution, or when it is ready to commit as in optimistic approach [5,7]. The version based algorithms [43] maintain several versions of database objects. Each update creates a new version. The version based algorithms are particularly useful for object oriented designs. Database implementors have tried all these approaches. Most systems have implemented the two-phase locking algorithms.

For time-stamp based concurrency control in a distributed database system there are two issues that require careful study: local vs. global transactions and atomic update of replicated copies.

2.3.1. Local vs. Global Transactions

In the optimistic concurrency control mechanism [7] a global transaction is validated against a set of previously committed transactions and is semi-committed on each site before a global commitment. Thus there is a time interval between the semi-commit and commit states of a transaction. For successful transactions, the updates on a site can be posted in the order in which transactions are semi-committed or globally committed on each site. However, local transaction requiring no access to distributed resources have no distinction between the semi-commit and commit states. If one implements the semi-commit order for updates, local transactions must wait until earlier global transactions are committed. If one implements the global commit order for updates the global transactions must revalidate again the local transactions that committed between its semi-commit and commit phases. A system could adapt between these two strategies in response to various mixes of local and global transactions.

2.3.2. Atomic Update of Replicated Copies

There is a time interval between the instance when a transaction commits and when its updates are posted on all copies. During this interval the concurrency controller must ensure that no other transaction reads the database that will be updated. Otherwise the concurrency controller will attempt to serialize such a transaction after the committed transaction even though based on its read actions, it should be serialized before the committed transaction. The key concern is to establish the correct *read-from* relation for each transaction in the distributed database system. We call this the *atomic update* problem. In locking based algorithms, since the transaction locks the database objects till the update is posted and then releases the locks for other transactions, this problem does not arise.

One of the solutions to the atomic update problem is that the transaction that update the database makes a record of its identification number (ID) on the database objects. Thus when a new transaction reads the database, it gets the ID of the transaction whose update it has read. This will establish a correct read-from relation that can be used during concurrency control. Another approach is to acquire global locks on all copies of the database objects at the beginning of the interval between the commit of concurrency control and the update of the database. We call these locks as *commit locks* since they are not the same as conventional locks used for concurrency control. These locks are short lived and are needed only for atomic updates of replicated copies.

We will identify the concurrency control mechanism used by different prototypes in Section 3. Since concurrency has been studied widely in literature, we avoid further details in this paper.

2.4. Replication

The database objects are replicated on different sites to increase availability. However, a replicated copy control (RC) mechanism is needed to ensure that all copies remain mutually consistent during transaction updates and failures.

A simple RC algorithm can be implemented to assure is that transactions may read any one copy but must update all copies or none of them. A variation of this is the majority consensus protocol where updating only a majority of copies is required [53]. In another approach the transaction is allowed to read more than one copy of a data object and the algorithm ensures that the sum of the number of copies read (read quorum) and the number of copies written (write quorum) is greater than the number of copies that exist [29]. This freedom is particularly important when one or more copies become unavailable due to site crashes or network partitions. To meet some of these constraints, commit algorithms have been designed [45]. The commit algorithms ensure that the transaction either writes to all required copies or none of them. These algorithms can involve one round (phase), two rounds, or three rounds of communication [45,49]. Most systems use a two-phase commit protocol. When failures are likely a three phase commit protocol should be used, to avoid blocking the progress of a transaction.

A variety of site failures can cause one or more copies of the database to become unavailable due to a crash or a communications failure (e.g., network partition). In a

system that requires that transactions read one copy and write-all copies, the failures can severely limit the availability of data for transaction processing. Recently an algorithm has been developed to deal with site failures [10,14]. It allows the transactions to read-any-copy and write-all-available copies. This idea has been implemented in the Locus and Raid system. The non-available copies are refreshed using *fail-locks* and *copier transactions*. Fail-locks are kept on the copies of data objects at operational sites to record the fact that some other copies have become out-of-date due to failures. During recovery the copies transactions refresh out-of-date copies by writing on them the values obtained reading an up-to-date copy. The overheads of fail-locks and copier transactions have been reported in [12]. Research is underway to deal with network partitions and multiple failures [10,24,25,31].

2.5. Heterogeneity

Existing database systems can be interconnected resulting in a heterogeneous distributed database system. Each site in such a system could be using different strategies for transaction management. For example, one site could be using the two phase locking concurrency control method while another could be running the optimistic method. Since it may not be possible to convert such different systems and algorithms to a homogeneous system, solutions must be found to deal with such heterogeneity. Already research has been done towards the designing of algorithms for performing concurrent updates in a heterogeneous environment [26,27]. The issues of global serializability and deadlock resolution have been solved. The approach in [27] is a variation of the optimistic concurrency control for global transactions while allowing individual sites to maintain their autonomy. Another concept that is being studied in the Raid system involves facilities to switch concurrency control methods. A formal model for an *adaptable* concurrency control [12] suggested three approaches for dealing with various system and transaction's states: generic state, converting state, and suffix sufficient state. The generic state method requires the development of a common data structure for all the ways to implement a particular concurrency controller (called sequencer). The converting state method works by invoking a conversion routine to change the state information as required by a different method. The suffix sufficient method requires switching from one method to another by overlapping the execution of both methods until certain termination conditions are satisfied.

Research is also underway on superdatabases and supertransactions for composition of heterogeneous databases [41]. In this approach, local databases produce explicit serial orders that are merged by the superdatabase into a global serial order. A number of successful prototype systems have been built [18, 28, 52].

3. IMPLEMENTATIONS OF PROTOTYPE SYSTEMS

In the last decade, many distributed database systems have been implemented to demonstrate the feasibility of concurrent transaction processing. Since 1980, reliability issues have drawn more attention. These deal with implementation of stable secondary storage, nonblocked processing during site failures and network partitions, maintenance of consistency among replicated copies, and implementation of reliable communication.

Several operating system kernels have been designed and extended to implement facilities for concurrent and reliable processing that are needed for database systems.

In this section, we briefly present eleven systems that have implemented many of the above facilities in database systems (Distributed Ingres, SDD-1, R*, Encompass, Sirius-Delta), in operating systems (Clouds, Eden, Camelot, Locus), in programming languages (Argus), or in communication software (Isis). This material was obtained from the published papers describing these systems or from the authors.

In the next section, we present in detail how these facilities have been incorporated in the distributed database system called Raid.

3.1. Distributed INGRES

Distributed Ingres consists of a process called *master Ingres* which runs at the site where the user's application is run. A process called *slave Ingres* runs at any site where data must be accessed. In addition, there is a receptor process which is spawned to accept relations which must be moved from one site to another. The master process parses the command, resolves any views, and creates an action plan for the query. The slave process is essentially a single machine Ingres with minor extensions and with the parser removed.

In 1979, a rudimentary version of distributed Ingres was operational at University of California, Berkeley on top of Vaxen running Unix. It provided for expansion of the number of sites in the system and partitioning of a database relation among the sites. In addition it allowed for distributed query processing. Distributed Ingres used the primary copy locking approach for concurrency control and a two phase commit protocol. Specialized networking software on top of Ethernet was developed for communications. Through several experiments, is was shown that distributed Ingres on a local database is about 20 percent slower than one site Ingres and on a remote site it is another 20 percent slower [49]. This project provided several research ideas in concurrency control, crash recovery and commit protocols, query optimization and networking.

3.2. R*

The experimental system R* was implemented by IBM in 1980. R* creates a process for each user on first request and retains that process for the duration of the user session. A virtual circuit and a server process are created on the first request of an R* process to an R* process at the server site. The requesting process and the serving process are bound by a virtual circuit. A request from a remote server process to a third site creates a new process and a new virtual circuit. Thus a user computation in R* comprises a tree of processes tied together by virtual circuits. R* sites communicate by using the SNALU6 protocol. The SNA virtual circuits are half-duplex circuits and can send a message of arbitrary length. The system uses locking based concurrency control and a nested form of the two phase commit protocol. Early performance measurements indicated that execution times for update-intensive transactions on remote data were 2 to 4 times slower than their local equivalents.

3.3. SDD-1

The distributed database management system called SDD-1 was built in 1978 by Computer Corporation of America. It is a collection of two types of servers: Transaction Modules (TMs), and Data modules (DMs) and a Reliable Network (RelNet). DMs are backend database systems that respond to messages from TMs which plan and control the execution of transactions.

SDD-1 uses the time-stamp mechanism for concurrency control and employs conflict graph analysis for identifying classes of transaction that require little or no synchronization. The database administrator defines the transaction classes based on the database objects that can be read or written. The system uses the idea of *semi-join* to reduce the transfer of data during distributed query processing. The RelNet provides for guaranteed delivery on ARPANET using separate spooler machines to record messages and send as necessary. The commit protocol is a variant of two phase commit with backup sites to store commit information. SDD-1 was implemented on DEC-10 and DEC-20 computers running the TENEX and TOPS-20 operating systems. The communication network was ARPA network. The research effort has lead to many ideas in timestamp based concurrency control methods, reliable communication, and distributed query processing.

3.4. ENCOMPASS

The Encompass distributed data management system is a commercial product of the Tandem Corporation. It provides a complete database management facility, terminal control, and transaction management. Transaction Monitoring Facility (TMF) is the name of the subsystem that is responsible for transaction mangement. TMF provides continuous, fault-tolerant processing in a decentralized distributed environment. TMF runs on the Tandem operating system [1] which is message based and is characterized by symmetry and the absence of any master/slave hierarchy. All communication is via messages. For reliability it uses the mechanism of process-pair. A process-pair consists of two cooperating processes which run in two processors physically connected to a particular device such as I/O.

The message-based structure of the operating system allows it to exert decentralized control over a local network of processors. The network provides automatic packet forwarding via an end-to-end protocol which ensures that data transmissions are reliably received.

The TMF uses locking methods for its concurrency control. Two types of granularities are provided; file level and record level. All locks are in exclusive mode. TMF uses an elaborate two-phase commit protocol for distributed transactions. This protocol allows any participating site to unilaterally abort a transaction. TMF maintains distributed audit trails of logical database record updates on mirrored disc volumes. The necessity for the ‘‘write ahead log’’ which requires before-images to be write-forced to the audit trail prior to performing any updates on disc have been eliminated. Instead the audit records are checkpointed to a backup prior to the update.

3.5. SIRIUS-DELTA

Sirius-Delta is a fully functional distributed system that has been running in INRIA, France since 1981. Its architecture has four layers. The lowest level manages the distributed execution functions and offers the services for remote process activation, job scheduling, inter-site concurrency management and data transfer among local actions. The next layer manages distributed control functions. This level uses a producer virtual ring, a control token, and a circulating sequencer to generate a unique and ordered set of names. At this level, two phase locking method is used for concurrency control and two phase commit for commitment. The next layer handles data distribution, and distributed query processing and optimization. The top layer runs the single site database system offered by ''Intertechnique'' that runs on Realite 2000 machines. Recovery is provided using a log and before-images.

3.6. LOCUS AND GENESIS

Locus is a Unix compatible, distributed operating system in operational use at University of California at Los Angeles on a network of 30 PCs, VAX/750s, and IBM 370s connected by a standard Ethernet since 1979. It supports transparent access to data through a network wide file system. Locus is a procedure based operating system. The processes request system services by executing system calls which trap to the kernel. The control structure is a special case of remote procedure calls. The operating system procedures are executed at a remote site as part of the service of a local system call.

For concurrency control, Locus enforces a global access synchronization policy for each filegroup. A given physical site can be assigned as the current synchronization site (CSS) for any file group. The CSS need not store any particular file in the filegroup but it must have knowledge of which sites store the file and what the most current version of the file is. Locking is used as the concurrency control method for this primary-copy update mechanism. The commitment mechanism requires committing the change to one copy of a file. The CSS blocks the changes to two different copies at the same time and prevents reading an old copy while another copy is being modified. As part of the commit operation, messages are sent to all copies and it is the responsibility of sites storing them to bring their versions up to date. The propagation of data is done by ''pulling'' rather than ''pushing'' the data.

Locus uses a shadow page mechanism for recovery. It provides treatment for network partitions via detection of inconsistencies due to updates, using a version number and version vector scheme. It implements a merge protocol to combine several subnetworks to a fully connected network. The replication copy mechanism of Locus is similar to the one used in Raid.

Genesis [38] is a distributed database operating system implemented as an extension of Locus. It is based on the premise that a considerable amount of distributed data management functionality could be obtained by simply running a single machine database system in a Locus environment that already provides remote data access, remote task execution, synchronization, recovery, replication, etc.

For example, a single machine Ingres, which was developed for Unix, runs on Locus. It accesses and updates remote data, has replicated relations, and works in the

face of network and site failures. In order to make use of the redundant processors, a remote subquery mechanism was added to Ingres. The system is able to run subqueries in parallel on any or all machines and to assemble the results.

In discussing the comparisons with distributed Ingres, it is stated that initial Genesis operational status was achieved by only two man months of additional programming and less than 1000 lines of source code to convert standard Ingres to run in parallel in a distributed environment. Performance data was obtained by running a simple query on a relation with 30,000 tuples, each 100 bytes long. The query took 2 minutes and 19 seconds (2:19) on Genesis Ingres with local data compared to 2:15 minutes on Ingres with local data. However, Genesis Ingres with remote data took 2:25 minutes compared to Ingres with remote data that took 3:23 minutes to execute.

The drawbacks of a distributed database on a system like Locus include increased I/O operations due to a small data block size and lack of a facility to cause pages of a file which are logically sequential to be stored in sequential order. Other arguments have been discussed in Section 4.6.

3.7. CLOUDS

Clouds is a distributed operating system implemented in 1986 at Georgia Institute of Technology. It is based on an object/process/action model. All instances of services, programs, and data in Clouds are objects. Processing is done via actions. Processes are carriers of the thread of control on behalf of actions. A Clouds object is an instance of an abstract data type. The object encapsulates permanent data and a set of routines that can access the data. Clouds supports passive objects as well as nested actions. Clouds uses passive objects to implement the procedure model. The code is not embedded in each process.

In Clouds, concurrency control and commitment are the responsibility of the objects touched by an action. The object compiler includes default code in the object entry and exit points to provide two phase locking. In addition to this automatic concurrency control, customized synchronization methods can also be implemented for many applications.

Like concurrency control, commitment methods have two types, namely custom and automatic. When an action is run with automatic concurrency control, the action management system keeps track of all the objects that are touched by the action. When the action completes its processing, the action management system creates a commit coordinator process. This process uses the pre-commit and commit entry points of all the objects and implements a two phase commit protocol. In the custom commit method, the programmer has the ability to redefine the default pre-commit and commit routines in the objects.

The Clouds system has its own implementation of a specially designed operating system kernel, in contrast to various other systems that have mostly used Unix. Clouds is designed to run on a set of general purpose computers that are connected via a medium to high speed local area network. Currently it runs on Vaxen connected via Ethernet. Clouds uses RPC with at-most-once semantics.

For distributed database applications [23], one can use an object for each relation. However, the code that accesses the data will be unnecessarily replicated. Clouds provides capability based access methods that allows common routines for accessing relations to be stored in a single object. Since it uses locking for concurrency control, the question of granularity becomes important. If each relation is an object, the granularity is too large. If each tuple becomes an object, the number of objects to be maintained by it could be in hundreds of thousands. Clouds allows for implementing a wide range of levels of granulary.

3.8. CAMELOT

Camelot is a distributed transaction facility that runs on top of the Mach operating system [42] at Carnegie Mellon University. It was implemented in 1987. It supports the execution of transactions and the implementation of abstract data objects. Camelot is based on the client-server model. It has implemented data servers which encapsulate shared recoverable objects. It uses remote procedure calls (RPCs) to provide communication among applications and servers. Camelot uses a combination of multi-threaded tasks, and shared memory to improve performance. Internet datagrams are used to coordinate transaction commitment and support distributed logging.

Camelot supports two types of concurrency control methods: locking and timestamps. Type-specific lock modes can be defined for higher concurrency. Write-ahead logging and checkpoints are used for implementing stable storage. Two options are provided for committing transactions: blocking commit (2 phases) and non-blocking commit (3 phases). A form of weighted voting is used for replication control.

Camelot implementation is an example of transaction and database support that can be built in lower level software. It takes advantage of facilities such as RPC, light weight processes (threads), and shared memory in Mach.

3.9. ARGUS

Argus is a programming language and system which supports the construction and execution of distributed programs and has been implemented at Massachusetts Institute of Technology. Argus allows computations to run as transactions. The database system can run as an application. This system has implemented support facilities for reliability, concurrency, and replication. Argus provides a linguistic mechanism called *guardian*. A guardian implements a number of procedures that are run response to remote requests. Guardians give programs a way of identifying units of tightly-coupled data and processing. A guardian encapsulates a set of objects and regulates access to these objects using handlers that can be called from other guardians.

In Argus, concurrency control and recovery are done through special objects called atomic objects. Two-phase locking is employed. For concurrency control, recovery in Argus is achieved by maintaining versions of objects. Guardians communicate with each other to perform a standard two-phase commit, in which modifications to an object are either moved from a volatile version to its stable version (to commit) or the volatile version is discarded (to abort).

Nested transactions [37] are an important concept used in the design of Argus. Many algorithms for concurrency control and recovery in nested transaction processing have resulted from this experimental effort.

The Argus implementation runs on micro VAX-IIs under Unix communicating over an Ethernet. The system was implemented in 1985. Argus has implemented several applications including a collaborative editor and a mail system.

3.10. EDEN

Eden is an object oriented distributed operating system implemented in 1982 at University of Washington. It consists of many homogeneous ‘‘node machines’’ (SUN workstations running Unix) connected by an Ethernet. It supplies two facilities for the Eden file system (EFS). The first is the abstraction of the Eden object, a dynamic entity, that in addition to type-defining code, can accommodate internal processes and internal communication. The second is a storage system with related primitives that allow the representation of an object to be checkpointed to secondary storage, to be copied, to be moved from node to node, and to be frozen. The EFS uses kernel primitives to store and retrieve objects and leaves the task of access to physical devices to the storage system.

Eden objects are defined by the user rather than being restricted to the system defined types. All communication among objects is through invocation which is a form of remote procedure. A high level Eden Programming Language (EPL) is used for defining Eden objects. It provides ‘‘light-weight’’ processes and monitors that the application programmer may use to control concurrency. EFS supports access to files under the control of a transaction manager (TM) object. Each TM controls one transaction. Thee configuration of EFS is very similar to that of SDD-1 system. The concurrency control method is based on version scheme. The versions are maintained by data managers (DMs). The two-phase commit protocol is used for commitment. Failure recovery is implemented via the use of a checkpoint mechanism. The voting scheme [29] is used for managing replicated copies. Replication control has been provided both at the kernel level and the object level. Recently nested transactions have been implemented successfully.

3.11. ISIS

Isis is an adjunct to a conventional operating system, providing persistent fault-tolerant services which complements conventional fault-tolerant programming support. The system was built in 1985 at Cornell University on a network of SUN workstations running Unix connected by an Ethernet.

Isis is an object oriented system that introduces a new programming abstraction called the ‘‘resilient object’’. Each resilient object provides services at a set of sites, where it is represented by ‘‘components’’ to which requests can be issued using remote procedure calls. The system itself is based on a small set of communication primitives which achieve high levels of concurrency. The communication system provides three types of broadcast primitives: BCAST, OBCAST, and GBCAST. The BCAST primitive ensures that if two broadcasts are received in some order at a common destination site,

they are received in the same order at all other common destinations. the OBCAST primitive ensures that multiple messages to a single destination are processed in the order they were sent. The GBCAST primitive transmits information about failures and recoveries to process groups. A virtually synchronous environment allows processes to be structured into ‘‘process groups’’, and provides for events like broadcast to the group as an entity.

The system is organized hierarchically. The lowest level provides communication primitives. Built on top of this is the layer supporting transactions and lock manager. On top of this layer are the broadcast primitives and the next level has fault-tolerant implementations that allow read, replicated write, commit and abort for transactions. Database applications can be built on this layer.

The Isis system uses two phase locking method for concurrency control. A read one, write-all copies rule is used when locking or accessing replicated data. For commitment the broadcast primitive employ variations of the two phase commit protocols. Recovery is accomplished using log based methods.

4. DESIGN AND IMPLEMENTATION OF RAID PROTOTYPE

Raid is a distributed database system that runs on a network of SUN workstations that run 4.3 BSD Unix interconnected by Ethernet. It has been operational since 1987 at Purdue University. It has a server based structure and uses messages for communications.

4.1. RAID Implementation Objectives

Raid has been implemented to support experiments that provide empirical evaluation of the design and implementation of algorithms for replicated copy control during failures, dynamic reconfiguration techniques, communications software, and transaction processing support in operating systems. These experiments identify principles that are necessary for reliable, high-performance transaction processing. An overview of the Raid project is in [13]. This implementation effort deals with some of the issues that are also being studied by many prototypes discussed in Section 3.

Raid provides replication control that is resilient to site and network failures. It uses a distributed two-phase commitment algorithm that makes use of validation concurrency control techniques to improve performance. Validation transmits the entire read/write-set in a single message, rather than requiring separate messages to lock each item. The distributed concurrency controller can use one of four different concurrency algorithms, as determined by the operator at run-time. The Raid communications subsystem provides location transparent virtual sites. Multiple virtual sites can run on the same physical host, permitting large-scale experiments on limited hardware. An oracle provides asynchronous notification of failures and recoveries.

The current version of Raid supports distributed transactions on a replicated database. A replication control subsystem provides support for failed sites to refresh their copy of the database and rejoin the system. RAIDTool provides a user-friendly operator interface.

4.2. RAID Site Structure

Raid provides complete support for transaction processing, including transparency to concurrent access, crash recovery, distribution of data, and atomicity. An instance of Raid can manage any number of virtual *sites* distributed among the available physical *hosts*.

Since reconfiguration is a principle goal of Raid, we chose to separate the modules that implement transaction processing into several UNIX processes, called servers. The servers communicate among themselves using the Raid communication package, which is layered on UDP/IP. The clean interface between servers simplifies incorporating new algorithms and facilities into Raid, or testing alternate implementations of algorithms. To permit concurrent transactions on a single site, there is a separate process for each user that coordinates the execution of her transactions with other servers. Since the Raid servers only communicate via our location-independent communication package, servers can be moved or replaced during the execution of transactions.

Figure 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, *etc*.

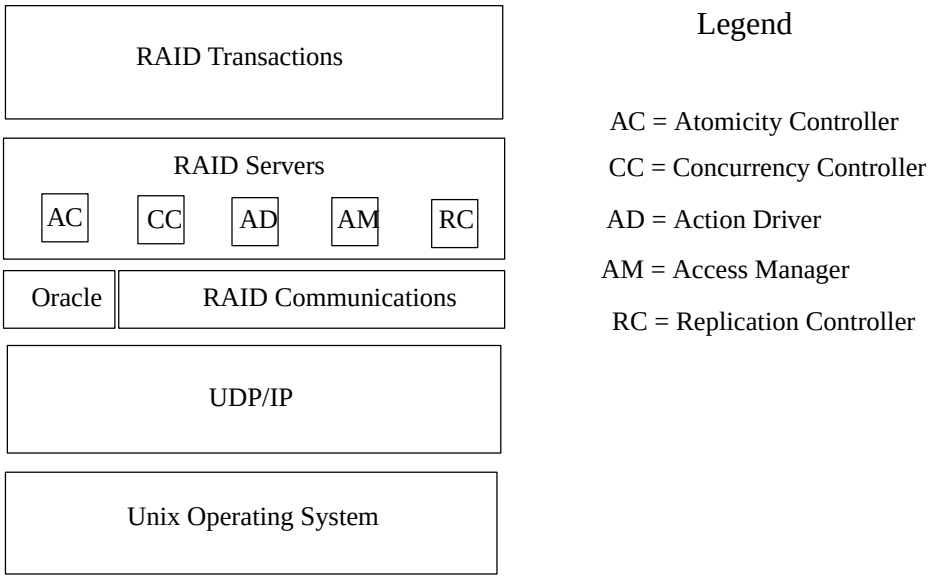

Figure 1: The organization of a Raid site.

Figure 2 shows the communication paths between the Raid servers.

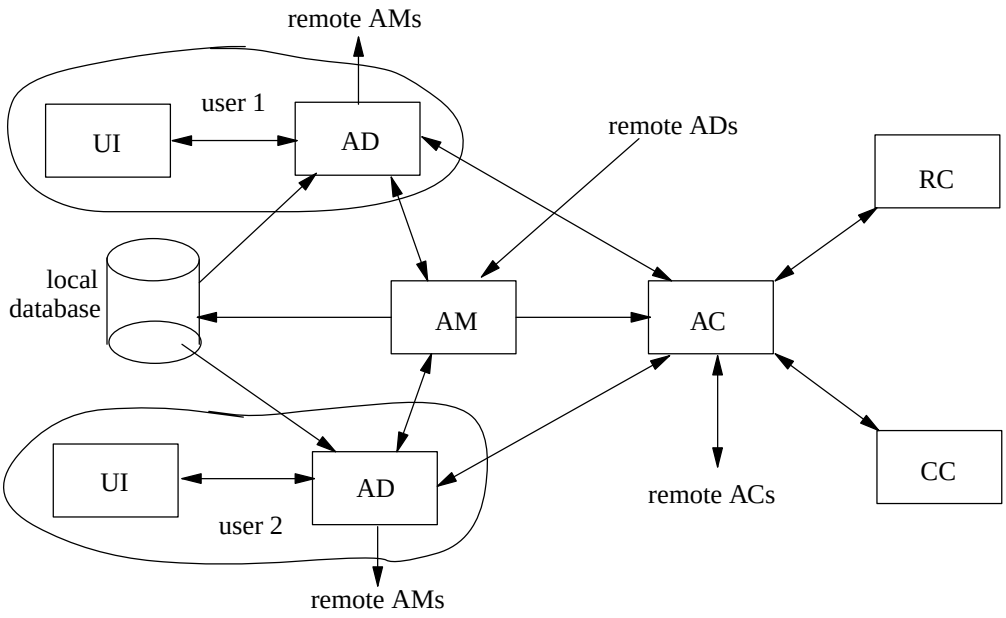

Figure 2: Communication structure of a Raid site.

The following describes the role of each of the Raid servers in the system.

4.2.1. User Interface

User Interface (UI) is a front-end invoked by a user to process relational calculus (QUEL-type) queries on a relational database. UI initiates its own Action Driver (AD), parses queries submitted by the user, and gives each parsed query to the AD. It then waits to hear from its AD whether the query committed or failed, and notifies the user of this fact.

4.2.2. Action Driver

Action Driver (AD) accepts a parsed query in the form of a tree of actions from its UI and executes the transaction, reading data from the local copy of the database. Transaction execution produces a transaction history that consists of read and write actions. After execution completes, the history is sent to Atomicity Controller (AC) to determine whether it is globally serializable. If the decision is ‘‘commit’’, data copies that need to be updated are sent to every Access Manager (AM) in the system.

4.2.3. Access Manager

Access Manager (AM) has two tasks. The first is to provide access to the relations, and to manage the indices and storage structures such as B-trees, K-D trees, etc. A Unix file is allocated to hold each relation.

The second task is to ensure that the updates of a transaction are written in a recoverable manner on stable storage. The AM serializes writes to the distributed database,

and communicates with the Raid commit mechanism. While a transaction is being executed by an AD, its updates are preserved either by a *log* that records the sequence of write actions, or in a *differential file* that contains new images of each relation modified by the transaction. When the transaction is ready to be committed, the differential file or log is passed by the AD to the AMs, which write it to stable storage. At this point the transaction is committed, since its updates will survive site failures.

4.2.4. Concurrency Controller

Concurrency Controller (CC) performs concurrency control using the time-stamp based validation approach [33]. It passes a transaction's history to all sites in the system, each of which validates whether the history is locally serializable with respect to previously successfully validated transactions on that site. Each local concurrency controller maintains information on globally committed transactions, and on previously locally validated transactions that are still in the commit process. If all sites report that the history is locally serializable, the transaction is globally serializable.

The concurrency controller in Raid has been implemented to perform the validation with one of many different implementations. Currently four different concurrency control methods have been implemented (simple locking, read/write locking, timestamping, and conflict graph cycle detection). The type of concurrency control may be chosen at run-time, but all transactions must be completed before it can be switched to a different type.

To ensure that either all or none of the updates on all copies take place, we use commit-locks [13]. Their implementation is discussed briefly in steps 5 and 11 of Section 4.4.

4.2.5. Replication Controller

Replication Controller (RC) maintains consistency among replicated copies and allows transaction processing as long as a single copy is available. The algorithm uses the read one-write all available (ROWAA) strategy discussed in [8,14]. The global status (which site is up and which site is down) of various sites is captured by a session vector whose elements represent the incarnation number of each site. Control transactions are used to signal changes in the session vector due to the failure or recovery of a site. Fail-locks are set on data objects that are updated while a copy is down and are used for refreshing copies when a site recovers.

4.3. RAID Communications

This section describes the high-level services provided by the Raid communications package, including the Raid name space, the oracle (name-server), and the available communications services.

4.3.1. The Name Space.

We can uniquely identify each Raid server with the tuple (Raid instance number, Raid virtual site number, server type, server instance). Figure 3 shows a possible

distribution of the servers among several physical hosts. Note that a single host can support multiple virtual sites, and that different servers of a single virtual site can reside on different hosts.

To send a message to a server, UDP needs a (machine name, port number) pair. The Raid oracle maps between Raid 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.

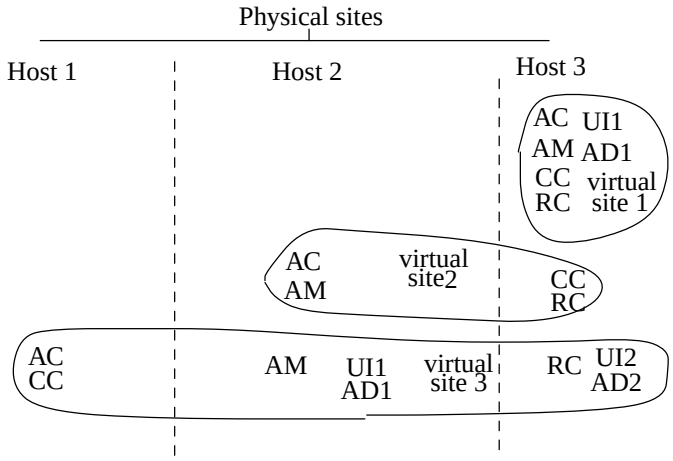

Figure 3: Possible distribution of Raid servers among physical hosts.

4.3.2. The Raid 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 *registration* and *lookup*. A server registers by telling the oracle its name (i.e., the tuple that uniquely identifies it). Any server can ask the oracle for the *lookup* service to determine the location of any other server.

Whenever a server changes status (e.g., moves, fails, or recovers) the oracle sends a notifier messsage to all other servers that have specified the changing server in their *notifier set*. Notifier messages are handled automatically by the communications software, which caches the address of the new server.

4.3.3. RAID Communications Facilities.

Raid servers communicate with each other using high-level operations. The high-level facilities are implemented on top of the low-level Raid transport protocol. We call this protocol *LDG*, for Long DataGram. This protocol is identical to UDP except that there is no restriction on packet sizes (many implementations of UDP restrict packet sizes to some maximum length). Each LDG packet is fragmented if necessary, and then sent using UDP. At the destination, fragments are collected and reassembled.

LDG is about twice the cost of UDP for packets that do not need to be fragmented. In this implementation, LDG datagrams are fragmented into 512 byte packets. Larger datagrams, which UDP transmits as a single packet, are much more expensive to transmit

using LDG.

4.4. RAID Transaction Processing

In order that the AC can manage multiple transactions simultaneously it is implemented in a multi-threaded manner. A multi-threaded 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.

0. UI gets a query from the user, parses it, and passes the parsed query on to AD.

1. AD assigns a globally unique transaction ID, and records the read-set and write-set as it processes the transaction using the local database. Updates are preserved in a log or a differential file.

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 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 the 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 the transaction's read-set. Copier transactions are generated for any out-of-date items that are found in the read-set.

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

5. If read-set is valid (no fail-locks), the AC acquires special commit-locks for the items in the write-set. 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 ACs. If the read-set is invalid, AC aborts this transaction.

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

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

8. AD sends the log or differential file to all AMs 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 was successfully written.

11. AC releases the commit-locks belonging to the transaction, and informs the local CC and all other ACs. The remote ACs release their commit-locks and inform their CCs. The CCs more 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.

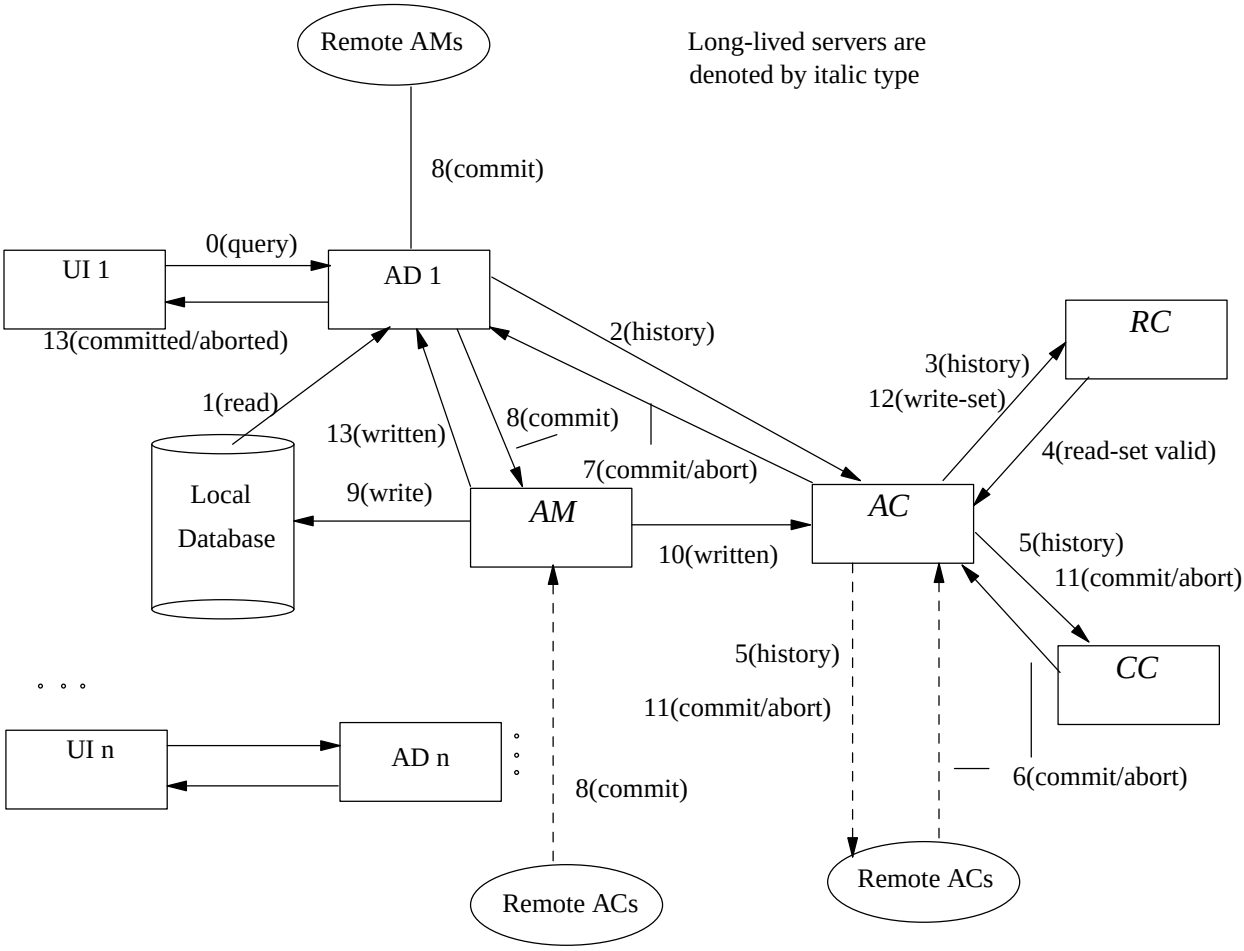

Figure 4: Transaction processing on a RAID site.

4.5. Summary of RAID Experience

Raid promotes experimentation through an organization that allows the substitution of one implementation for another at run-time. Central to this capability is the server philosophy, which implements each functional component of the system in a separate operating system process. These servers communicate through a high-level message service. The messages exchanged between servers constitute a clearly defined interface to each component of the system. New implementations need only match this interface to run correctly as Raid servers.

The implementation has proven satisfactory for an experimental system, but several of our design decisions have influenced performance in a negative manner. The use of

ASCII as the network data type has been helpful for debugging, but the cost of converting numeric data to and from ASCII is higher than we would like. This cost is most significant in our LUDP implementation, since even the fragment headers are converted to ASCII. Converting a single number can take as long as 200 μ seconds, but even more important is the difficulty of supporting variable length packet headers. A significant amount of time is wasted allocating and deallocating buffers of different sizes.

The need for concurrent execution in several server types has been an implementation problem. The Action Driver 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 Atomicity Controller 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 could help with this scheduling problem. Each of these servers would benefit from operating system support for lightweight processes [20,42]. Lightweight processes are easy to create and destroy, and their context switches are inexpensive, because a group of them resides in a single, shared address space. Creating a lightweight process does not require allocating a page table, deleting it does not require deallocating memory, and context switching between two lightweight processes in the same address space avoids the overhead of changing page tables and flushing memory caches.

4.5.1. Enhanced Operating System Services in the RAID System

Operating system support for the processing of database transactions has been studied by both database and operating systems researchers. The limitations of conventional operating system services for a database have been presented in [16,48,50]. In particular, Unix is criticized in the areas of buffer pool management, process scheduling, file system performance, and support for recovery. In each of these areas improved services would facilitate the implementation of high performance, reliable database systems. An alternative view argues that most of the Unix facilities *are* suitable for database processing [54]. Simple modifications to the file system and to the cache management can improve performance significantly. For example, the Unix fork call can be modified to avoid copying memory pages that are not modified in the new process. Since fork is often used immediately before **exec**, which completely replaces the memory image of the calling routine, the new implementation often avoids copying altogether. Using this lightweight process mechanism, the speed of processing for small transactions is doubled.

The close relationship between the transaction processing system called Camelot [46,47] and an operating system called Mach [42] has led to the development of a very interesting set of operating system services. Camelot relies on efficient inter-process communication and local synchronization provided by the Mach system [42].

We now discuss the potential use of enhanced operating system services within Raid. They represent Unix weaknesses in inter-process communication both within a single host and among hosts.

When Raid is processing distributed transactions, it uses a multi-phase commit procedure to ensure atomicity. Each phase of the commit protocol consists of the master site that broadcasts a request to all other sites and then waits for replies from each of them. Low-level communications support for multicast would decrease the commit delay, which in turn, would increase the number of transactions that are able to commit, since conflicts would have less time to occur. Optimally, multicast would use an available hardware multicast mechanisms, but even its simulation in the software at the kernel level is useful. Our Ethernet experiments [9] demonstrated that a substantial part of datagram latency can be attributed to getting in and out of the kernel. Simulated multicast in the kernel would incur this cost only once per packet, regardless of the number of destination hosts.

The Raid communications package provides high-level simulated broadcast calls. These calls can be modified to use the known lower-level multicast protocols (e.g., VMTP [19]) for improved efficiency. This can be done by modifying the SE system call that has been described in [9].

4.5.2. Shared Memory Facilities

Systems designers are tempted to build required services into the operating system kernel rather than in user-level code. One reason is to take advantage of improved efficiency by making use of the single kernel address space. User-level shared memory facilities can help reduce this temptation by supporting efficient implementation of inter-process communication at the user level.

The simplest method of using these services is to replace the LUDP datagram service of the communications package with a new service that copies the message to a shared memory segment, if the destination is on the same host as the source. This approach avoids the overhead of a kernel system call, does not require the copying data in and out of the kernel, and allows for the sending of very large datagrams with a single procedure call. The necessary synchronization requires a semaphore call to the kernel, but this is likely to be more efficient than sending a message via the kernel. Such shared memory and synchronization services are provided in the release 3.2 of SUN Unix.

5. CONCLUSION

A lot of theoretical work has been done in the design of distributed database systems and transaction processing. Many concepts models, and algorithms have been developed for concurrency control, crash recovery, communication, query optimization, nested transactions, site failures, network partitions, and replication. Simultaneously a major effort has been put in the implementation of these ideas in experimental and commercial systems with a wide spectrum of distributed database systems, distributed operating systems and programming languages. The implementors have discarded many ideas while adopting others such as two phase locking, two phase commit, and remote procedure calls. They have built workable systems but still wrestle with the problems of

reliability and replication. Performance issues are of major concern as we move towards applications that process 1000 transactions per second in a reliable manner. Research continues in the area of transaction processing on replicated data when site failures and network partitions can occur. The true benefi ts of distributed database systems such as high availability and good response time appear to be within reach. Further experimentation on the relationship between operating systems and database systems, on the impact of replication on availability and performance during failures, on the effects of scaling (increasing the number of sites) on transaction throughput and reliability, and on designs for reliable communication software with reduced delays will lead to better systems and confi dence in the capabilities of distributed systems

6. ACKNOWLEDGEMENTS

I thank Professor Ahmed Elmagarmid for inviting me to write this paper. I also thank the members of the Raid project and in particular John Riedl for discussions on the design and implementation issues.

7. REFERENCES

1. Bartlett, J., ‘‘A NonStop Kernel’’, in *Proceedings of the Eighth Symposium on Operating System Principles*, pages 22-29, Pacifi c Grove, California, December 1981.

2. Bernstein, P., V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Co., Reading, MA, 1987.

3. Bernstein, P. and N. Goodman, ‘‘Concurrency Control in Distributed Database Systems’’, *ACM Comput. Surveys*, Vo. 13, pp. 185-221, June 1981.

4. Bernstein, P., D. Shipman, and J. Rothnie, Jr., ‘‘Concurrency Control in a System for Distributed Databases (SDD-1)’’, *ACM Trans. on Database Systems*, vol. 5, pp. 18-25,

5. Bhargava, B., ‘‘Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking’’, in *Proceedings of 3rd IEEE Distributed Computing Systems Conference*, Fort Lauderdale, FL, Oct. 1982.

6. Bhargava, B., Editor, *Concurrency and Reliability in Distributed systems*, Van Nostrand and Reinhold, 1987.

7. Bhargava, B., ‘‘Resilient Concurrency Control in Distributed Database Systems’’, *IEEE Transactions on Reliability*, vol R-32, No.5 (1983), 437-443.

8. Bhargava, B., ‘‘Transaction Processing and Consistency Control of Replicated Copies During Failures’’, *Journal of Management Information Systems*, Vol 4, No. 2 (1987) 93-112.

9. Bhargava, B., T. Mueller, and J. Riedl, ‘‘Experimental Analysis of Layered Ethernet Software’’, in *Proceedings of the ACM-IEEE Computer Society 1987 Fall Joint Computer Conference*, Dallas, Texas, October 1987, pages 559-568.

10. Bhargava, B. and P. Ng, ‘‘A Dynamic Majority Determination Algorithm for Reconfiguration of Network Partitions’’, to appear *International Journal of Information Science*, special issue on Database Systems, February 1988.

11. Bhargava, B., P. Noll, and D. Sabo, ‘‘An Experimental Analysis of Replicated Copy Control During Site Failure and Recovery’’, in *Proceedings of the Fourth IEEE Data Engineering Conference*, February 1988, Los Angeles, California (available as CSD-TR-692).

12. Bhargava, B. and J. Riedl, ‘‘A Formal Model for Adaptable Systems for Transaction Processing’’, in *Procededings of the Fourth IEEE Data Engineering Conference*, February 1988, Los Angeles, California (available as CSD-TR-609).

13. Bhargava, B., J. Riedl, A. Royappa, ‘‘RAID Distributed Database System’’, Technical Report CSD-TR-691, Purdue University, Aug. 1987.

14. Bhargava, B. and Z. Ruan, ‘‘Site Recovery in Replicated Distributed Database Systems’’, in *Proceedings of the Sixth IEEE Distributed Computing Systems Conference*, pages 621-627, May 1986.

15. Birman, K., ‘‘Replication and Availability in the ISIS System’’, *Operating System Review*, 19(5):79-86, December 1985.

16. Boral, H., *Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, December 1986, (special issue on Operating System Support for Database Systems).

17. Borr, A., ‘‘Transaction Monitoring in Encompass’’, in *Proceedings of the 7th Conference on Very Large Databases*, September 1981.

18. Ceri, S. and G. Pelagatti, *Distributed Databases Principles and Systems*, McGraw-Hill Book Co., New York, NY, 1984.

19. Cheriton, D., ‘‘VMTP: A Transport Protocol for the Next Generation of Communication Systems’’, in *SIGCOMM ’86 Symposium*, pages 406-415, ACM, August

1986.

20. Cheriton, D. and W. Zwaenepoel, ‘‘Distributed Process Groups in the V Kernel’’, *ACM Trans. Comput. Sys.*, 3(2):77-107, May 1985.

21. Comer, D., *Operating System Design, Volume II. Internetworking with XINU*, Prentice-Hall, Inc. 1987.

22. Dasgupta, P., R. LeBlanc, and E. Spafford, ‘‘The Clouds Project: Design and Implementation of a Fault Tolerant Distributed Operating System’’, Technical Report GIT-ICS-85/29, Georgia Institute of Technology, Oct. 1985.

23. Dasgupta, P. and M. Morsi, ‘‘An Object-Based Distributed Database System Supported on the Clouds Operating System’’, Technical Report GIT-ICS-86/07, Georgia Institute of Technology, 1986.

24. Davcev, D. and W. Burkhard, ‘‘Consistency and Recovery Control for Replicated Files’’, in *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 87-96, Orcas Island, Washington, December 1985.

25. Davidson, S., H. Garcia-Molina, and D. Skeen, ‘‘Consistency in Partitioned Networks’’, *ACM Computing Surveys*, 17(3), September 1985.

26. Elmagarmid, A. and A. Helal, ‘‘Supporting Updates in Heterogeneous Distributed Database’’, in *Proceedings of Fourth IEEE Data Engineering Conference*, Los Angele, Feb. 1988.

27. Elmagarmid, A. and Y. Leu, ‘‘An Optimistic Concurrency Control Algorithm for Heterogeneous Distributed Database System’’, *Data Engineering*, Vol. 10, No. 3, Sept. 1987, pp. 26-32.

28. Ferrier, A. and C. Stangret, ‘‘Heterogeneity in the Distributed Database Management System SIRIUS-DELTA’’, *Eighth VLDB*, Mexico City, 1983.

29. Gifford, D., ‘‘Weighted Voting for Replicated Data’’, in *Proc. of the 7th ACM Symp. on Operating System Principles*, pp. 150-159, Pacifi c Grove, California Dec. 1979.

30. Gray, J., ‘‘Notes on Database Operating Systems’’, in *Lecture Notes in Computer Science 60, Operating Systems*, and Advanced Course, ed. R. Bayer, et. al., pp 398-481, Springer Verlag, New York, New York, 1978.

31. Jajodia, S., ‘‘Managing Replicated Files in Partitioned Distributed Database Systems’’, in *Proceedings of the IEEE Data Engineering Conference*, Los Angeles, California, 1987.

32. Jessop, W., J. Noe, D. Jacobson, J. Baer, C. Pu, ‘‘The EDEN Transaction-Based File System’’, in *Proceedings of Second IEEE Computer Society’s Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, Pennsylvania, 1982, pp. 163-169.

33. Lamport, L., ‘‘Time Clocks, and the Ordering of Events in a Distributed System’’, *Communications of ACM*, vol. 21, pp. 558-564, July 1978.

34. Leu, P. and B. Bhargava, ‘‘Multidimesional Timstamp Protocols for Concurrency Control’’, *IEEE Transactions on Software Engineering*, SE-13, No. 12 (1987) 1238-1253.

35. Lindsay, B., L. Haas, C. Mohan, P. Wilms, and R. Yost, ‘‘Computation and Communication in R*: A Distributed Database Manager’’, *ACM Trans. Comput. Sys.*, 2(1), February 1984.

36. Liskov, B., D. Curtis, P. Johnson and R. Scheifler, ‘‘Implementation of Argus’’, in *Proceedings of the 11th ACM Symposium on Operationg Systems Principles*, Nov. 1987.

37. Moss, J., ‘‘Nested Transactions: An Introduction’’, *Concurrency Control and Reliability in Distributed Systems*, Chapt. 14, (B. Bhargava, ed.), pp. 395-425, 1987.

38. Page, T., Jr., M. Weinstein, and G. Popek, ‘‘Genesis: A Distributed Database Operating System’’, in *Proceedings of ACM SIGMOD Conference, Austin, Texas, 1985, pp 374-387.*

39. Papadimitriou, C., *The Theory of Concurrency Control*, Computer Science Press, Rockville, MD, 1986.

40. Popek, G. and B. Walker, (ed.) *The LOCUS Distributed System Architecture*, MIT Press, 1985.

41. Pu, C., ‘‘Superdatabases for Composition of Heterogeneous Databases’’, Technical Report CUCS-243-86, Columbia University, Dec. 1986.

42. Rashid, R., ‘‘Threads of a New System’’, *Unix Reveiw*, 4(8):37-49, August 1986.

43. Reed, D., ‘‘Implementing Atomic Actions on Decentralized Data’’, *ACM Trans. Computer Syst.*, vol. 1, pp. 3-23, Feb. 1983

44. Rothnie, J., Jr., P. Bernstein, S. Fox, N. Goodman, M. Hammer, T. Landers, C. Reeve, D. Shipman, E. Wong, ‘‘Introduction to a System for Distributed Databases (SDD-1)’’, *ACM Trans. on Database Systems* 5(1):1-17, March 1980.

45. Skeen, D. and M. Stonebraker, ‘‘A Formal Model of Crash Recovery in a Distributed System, *IEEE Trans. on Software Engineering*, vol. SE-9, no. 3 (1983).

46. Spector, A., D. Thompson, R. Pausch, J. Eppinger, D. Duchamp, R. Draves, D. Daniels, and J. Block, *Camelot: A Distributed Transaction Facility for MACH and the Internet – An Interim Report*, Technical Report CMU-CS-87-129, Carnegie Mellon University, June 1987.

47. Spector, A.Z., J.J. Bloch, D.S. Daniels, R.P. Draves, D. Duchamp, J.L. Eppinger, S.G. Menees, and D.S. Thompson, ‘‘The Camelot Project’’, *Database Engineering*, 9(4), December 1986.

48. Stonebraker, M., ‘‘Operating System Support for Database Management’’, *Commun. ACM*, 24(7):412-418, July 1981.

49. Stonebraker, M., (ed.), *The INGRES Papers*, Addison-Wesley Publishing Compay, 1986.

50. Stonebraker, M., ‘‘Problems in Supporting Data Base Transactions in an Operating System Transaction Manager’’, *Operating System Review*, January 1985.

51. Tanenbaum, A. and R. Renesse, ‘‘Reliability Issues in Distributed Operating Systems’’, in *Proceedings of Sixth IEEE Symposium on Reliability in Distributed Software and Database Systems*, Williamsburgh, Virginia, 1987, pp. 3-11.

52. Templeton, M., et. al., ‘‘Mermaid-Experiences with Network Operation’’, in *Proceedings of the Second IEEE Data Engineering Conference*, Feb. 1986, pp. 292-300.

53. Thomas, R., ‘‘A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases’’, *ACM Trans. Database Syst.*, vol. 4, pp. 180-209, June 1979.

54. Weinberger, P.J., ‘‘Making UNIX Operating Systems Safe for Databases’’, *The Bell System Technical Journal*, 61(9), November 1982.