Genesis: A Distributed Database Operating System

Thomas W. Page, Jr., Matthew J. Weinstein, and Gerald J. Popek

University of California, Los Angeles

Abstract

An approach to the architecture of distributed databases is presented, oriented around extensive embedding of database primitives in the underlying distributed operating system. It is shown that this approach greatly reduces the effort required to provide high quality distributed database services. Substantial portions of this approach are implemented and are operational as extensions to the Locus distributed Unix operating system. This paper describes the approach, compares it with other work, argues why it is attractive and what drawbacks are present, and lays out what has been done and what remains.

1 Introduction

The plummeting cost and increasing power of individual workstations, together with the wide availability of low cost, high speed local area networks, and improved computer communications generally, greatly increase the desire for integrated, transparent distributed database systems. The interface to the distributed database should be transparent That is, the system should provide support by which machine boundaries are functionally invisible, in the same manner that virtual memory hides the boundary between primary and secondary storage

This research has been supported by the Advanced Research Projects Agency Under contract DSS-MDA-903-82-C-0189

The question immediately arises at what level within the system architecture tranparency support should be provided. This issue is especially important for distributed data management, because of the preeminence of this application. The traditional approach to distributed data management uses existing single machine computing systems, connected by conventional long haul networking techniques such as TCP/IP, SNA or ISO protocols as the base. In these environments, it is the database system developer's responsibility to provide for remote data access, creation and control of remote task execution, implementation of distributed transactions and their synchronization and recovery, etc. Because the application level implements these functions, it typically is difficult or impossible to share them among different database systems, or between a database system and a front end transaction processing facility, or even between these packages and user programs written as extensions.

As a result, these functions are implemented multiple times, once in each application. This practice is expensive, representing many man years of duplicate effort, and also does not effectively address the desire to couple the mechanisms in the several implementations (e.g. compose an atomic, synchronized transaction out of a user program, a transaction processor (TP) execution, and a database update). Clearly support for database functions in the common operating system base would be a boon.

Nevertheless, database developers often explicitly prefer to bypass the underlying operating system, since operating systems generally do not give applications software sufficient control over resources. For normal, untrusted applications, that lack of control is intended to block one application from substantially degrading the behavior of others. However, the database system typically knows a great deal about its client behavior, represents a

substantial portion of system resource consumption, and generally is to be trusted. Hence, the supporting operating system should provide methods by which such a favored application can directly manage system resources itself, rather than requiring it to reimplement similar facilities in order to achieve that control. If the operating system can be altered to provide such methods, then this problem need not be overcome at such great expense, and the basic operating system can be used directly without penalty.

It is also worth noting that for many distributed environments, the network is likely to be a high bandwidth local area network of some kind, rather than a much slower and higher delay long haul medium. This fact permits one to consider treating the network more like a buss connecting processors in a multiprocessor system than a network using an "arms length" long haul protocol. While shared memory speeds are by no means available over such networks, still full transparency may well be achievable.

Based on these views, the Genesis distributed database system architecture was developed, and prototype implementations constructed. The implementation approach taken was to start with the Locus distributed Unix operating system [Walker83b] and the single machine Ingres [Stonebraker76] database system. Both Locus and Ingres were modified. The experience of using these two systems to demonstrate the suitability of the Genesis approach was very positive This paper reports some of the results in this ongoing distributed database work.

In the next sections, we first briefly review other distributed database system research. Then a discussion of the use of Locus and Ingres is presented. The Genesis work which is already done is reported, as well as that which remains. Conclusions follow.

2 Related Research

While there is a large body of literature concerning distributed databases, there are relatively few substantial implementations. Some of the most significant projects are reviewed here. It is important to note that, with the partial exception of Tandem/ENCOMPASS, these systems were not built on top of distributed operating systems. They implemented all of those mechanisms dealing with the distributed nature of the data and processors within the database.

2.1 Distributed Ingres

The University of California, Berkeley worked toward a distributed version of Ingres [Stonebraker77] [Stonebraker83]. Many features of this system, including a simple distributed query optimizer [Epstein78] were operational on a collection of VAX 11/780 and 11/750s connected by an Ethernet. The database ran on top of the 4.1cBSD version of Unix, using the remote interprocess communication and remote process execution facilities provided by that system to communicate among machines.

The system architecture consisted of a master Ingres process at the site at which a query originated, and slave processes at each of the sites storing data involved in the query. The slave processes were similar to single machine Ingres images without a parser [Stonebraker76]. The master Ingres parsed the query, devised an access plan and caused the plan to be carried out by the slave processes. Distributed Ingres implemented a two-phase commit protocol to post updates to the database atomically.

2.2 SDD-1: Computer Corporation of America

SDD-1 was among the first general purpose distributed data management systems built [Rothnie80]. It ran on DEC-10 and DEC-20 computers using the TENEX and TOPS-20 operating systems. SDD-1 used the ARPA network to connect machines and consequently assumed that communications were a bottleneck.

The SDD-1 architecture consisted of a collection of three types of virtual machines: Transaction Modules (TMs), Data Modules (DMs) and a Reliable Network. Data was managed by the DMs, which were essentially back-end single site databases resembling the slave processes in the Ingres architecture. The DMs responded to commands from the TMs which controlled the distributed execution of transactions. Concurrency control was accomplished using a timestamp algorithm (as opposed to locking) implemented in the TM.

2.3 The R* Distributed Database

R* is an adaptation and extension of the System R relational database system developed by IBM San Jose [Haas82]. It runs on IBM/370 family systems, using the inter-machine facilities of CICS.

R* is composed of four major subsystems: the storage subsystem, communication component, transaction manager, and database language processor. The database language processor compiles queries, performing extensive optimization based on both message and processing costs. At runtime, queries are initiated at appropriate slave processors by the transaction manager, which coordinates the multi-site transactions using the communication component, and recovery mechanisms and concurrency controls provided by the storage manager.

2.4 Tandem/ENCOMPASS

ENCOMPASS is a database/transaction manager that runs on Tandem computer systems [Borr81]. ENCOMPASS takes advantage of reliability features present in the Tandem hardware and software systems, such as disk volume mirroring and primary/backup process pairs, to provide a high-throughput transaction environment.

ENCOMPASS is composed of four functional components, including a database manager, a transaction manager, and a distributed transaction manager. The latter two compose the Transaction Monitoring Facility (TMF) of ENCOMPASS, which provides lock management, log management, and recovery management services to database applications. The underlying communication service used by applications to communicate with the TMF is a message-passing mechanism supported by the Tandem hardware and operating system software.

3 Distributed Databases in Genesis

The Locus system [Popek81] reflects our view that the operating system should support those mechanisms dealing with distributed operation which are applicable to many different client applications. Consequently, many of the facilities which distributed databases require are already present in Locus, a distributed operating system providing a very high degree of network transparency. A Locus network appears to users and application programs like a single computer running Unix. In order to provide network transparency and reliable operation, Locus handles concurrent access to files, distributed directory management, replication, transparent remote tasking, network interprocess communications, partitioned operation, and recovery from failures [Parker81] [Walker83a]. These are among the functions distributed database systems require in order to present a transparent view of the network database.

As Locus provides many of the mechanisms that distributed databases must have, one expects that a considerable amount of distributed data management functionality could be obtained simply by running a single machine database system in a Locus environment. In fact, since Locus is application code compatible with Unix, any database that runs under Unix will become trivially distributed on a Locus network. For example, single machine Ingres, which was developed for Unix, runs on Locus without modification. It accesses and updates remote data, has replicated relations (files), and works in the face of network and site failures.

While a single site database in Locus accesses remote data, it is a distributed database in a limited sense. Not all of the potential benefits of the network are harnessed. All data is brought across the network for for local processing. The many other processors in the system are wasted. Not only could the use of remote processors speed the query by operating on portions of the job in parallel, but it could also reduce the amount of data transmitted over the network. For many important cases, centralized systems like Ingres or IMS are cpu bound [Hawthorne79]. To alleviate this problem, it is important to make use of the redundant machines to process portions of the query in parallel. In order to make use of the redundant processors, the single site database must be enhanced with a remote subquery processing mechanism.

3.1 A Case Study: Genesis Ingres

The Genesis approach to the architecture of a distributed database is oriented around embedding database primitives in the distributed operating system. In order to demonstrate the Genesis architecture, we modified Ingres to take advantage of the facilities in a Locus network [Page85]. A remote subquery mechanism was added to Ingres which gives the database the option of bringing the

data to the process or moving the processing to the data storage sites. The system is able to run subqueries in parallel on any or all machines and to assemble the results. This mechanism makes use of remote query server processes on each site. A master Ingres process running on the user's site determines the storage site for all data involved in a query and may send query fragments to the server Ingres processes on remote sites.

3.1.1 Comparison With Berkeley Distributed Ingres

The master and slave processes resemble the design used in the Berkeley Distributed Ingres project. However, the two systems differ dramatically in their basic architecture. Distributed Ingres, built on top of a network of computers each running Unix, had an explicit view of the underlying distributed system. Consequently the mechanisms to deal with the distributed nature of the data and processing resources resided in the database system By contrast, except for the remote subquery mechanism, Genesis Ingres treats the underlying system as a single machine. While valid performance comparisons between Berkeley Distributed Ingres and Genesis Ingres are not yet possible as both systems are in their infancy, initial comparisons strongly support the Genesis approach. Real comparisons *can* be made of the way in which the two systems are constructed. The UCLA implementation, built in a network transparent environment, required only two man months of programming and less than 1000 lines of source code to convert standard Ingres to run in parallel in a distributed environment. The Berkeley effort on top of the conventional Unix environment, by contrast, took about four man years of programming effort to get an initial system up. While neither package is a fully functioning distributed database, the time required to get an initial system operational is very important, as further enhancements may be made and tested incrementally. That a system exhibiting reasonable performance can be built on top of a network transparent distributed operating system in so short a time is a significant result.

3.1.2 Performance of Genesis Ingres

Timing measurements illustrate the performance of the remote subquery mechanism. The following query was run on the version of Locus operational at UCLA in December 1983:

```
RANGE OF t IS test
RETRIEVE (t.all) WHERE (t number < 0)
```

The network consisted of 21 Vax 11/750s with Fujitsu Eagle disks connected by a 10 Mb/s Ethernet.

The relation "test" contained 30,000 tuples, each 100 bytes long, so the relation contained a total of 3MB of data. No tuple had a "number" field less that 0 so no tuples satisfied the qualification in the query. The table below shows the elapsed time to execute this query when run in five different environments. The first line shows the time required for single site (Berkeley) Ingres to run on a single Vax 11/750. The second result is for our enhanced distributed Genesis Ingres, but running on a Locus network with all the data local to the query site (i.e. not distributed at all). Next is shown the result for standard Ingres running in Locus with the data stored at a single remote site. In this case the database uses the network transparent file system to bring the data to the execution site. Line four shows Genesis Ingres retrieving remote data by running the entire query as a subquery at a the remote storage site. The last result, the real distributed case, shows the performance of distributed Genesis Ingres when the data is partitioned equally on each of three remote machines and processed in parallel by subqueries on the storage sites.

	min.
Ingres with local data	**2:15**
Genesis Ingres with local data	**2:19**
Ingres with remote data	**3:23**
Genesis Ingres remote data	**2:25**
Genesis Ingres data on 3 sites	**:50**

Table 1: Performance of different Ingres architectures

These results demonstrate the potential of our architecture. It takes only slightly longer to run Genesis Ingres for local data than to run standard Ingres, i e. the overhead of our mechanism is not significant. The time required to run the stan-

dard database on remote data is about one minute eight seconds longer than for local data. This is quite reasonable, since the time required to read a 3MB file across the network in the version of Locus used is about one minute five seconds. Running Genesis Ingres on remote data with a remote subquery run at the storage site takes about the same amount of time as did the first two trials. Genesis Ingres with remote data is about one minute faster than is standard Ingres, because the distributed version is able to run at the data storage site, avoiding any data being sent over the network. Most significant, retrieving the data in parallel from three sites runs as hoped in about one third of the time that retrieving the entire relation from one site does. This final result is very encouraging as it implies that for large queries, the relative overhead of the remote subquery mechanism is not substantial and we do achieve the performance gains of parallel operation.

This initial work demonstrates the advantage of the proposed database architecture from a software engineering point of view. Early measurements indicate that the benefits of implementing data management support in an underlying distributed operating system need not be at the expense of database performance. We now address some of the arguments against implementing these facilities in the operating system.

3.2 Criticisms of a Distributed Database Operating System

It is important to address the common objections to building a distributed database on top of a distributed operating system, and on top of Genesis in particular Such an approach, as mentioned earlier is often criticized on the basis that an operating system is built for various applications and is not optimized for database requirements. Such criticism is based on the assumption that the database designers must take the operating system as a given. By contrast, in building a distributed database operating system, the designers have the same opportunity as do database developers to optimize the basic system for data management applications If a given operating system provides the wrong service or performs that service poorly, it should not be concluded that the facility must be rebuilt in the database, but rather that it should have been done correctly in the operating system.

More importantly, a database system is not so much itself an application, but one of the programming tools that an application uses If the mechanisms to provide transparency, reliability and good performance in a network environment are not implemented in the underlying operating system, but instead in the database software, only resources controlled by the database can make use of these facilities. A transaction processing system, for example, which makes use of a database for part of its work will wish to access remote files, devices, or processors and would benefit from transparent access to these resources. As we see in the next section, higher level applications require that atomic transactions, which are usually provided by a database system, work across both database and non-database objects. In order to coordinate concurrent transactions, a distributed locking mechanism must be constructed for non-database as well as database objects. Given that many facilities which are expensive to construct must be built to make a distributed database network transparent, reliable, and efficient, there is great motivation to implement these features in the operating system and consequently make them available to all applications.

The Unix (and consequently Locus) file systems have several features which are criticized as limiting the performance of database systems (see for example [Stonebraker 81]. The small data block size increases the number of I/O operations that must be performed to retrieve or modify data. There is no facility to cause pages of a file which are logically sequential to be physically stored in sequential order; hence the Unix file system does not achieve the potential performance gains that result from the sequential behavior typically exhibited by database queries. Finally, cpu time is spent by the system to first read pages into system space and then copy them into the user space accessible to the database.

In our judgement, these are valid criticisms, not of the approach of constructing a data management system on top of an operating system, but rather of the current implementation of the Unix file system. The modifications made to the file system in the Berkeley 4.2 release of Unix address the block size and physical contiguity problems. The relatively few extra instructions required to copy data from kernel to user space generally pale when compared to the thousands of instruc-

tions required to retrieve pages from local or remote disks.

In sum, Locus appears to be a good basis upon which to build the Genesis distributed database operating system. It already provides atomic commit, automatic updates to distributed replicated files, a network transparent name service, remote tasking and inter-process communications and automatically brings out of date copies of replicated data up to date after recovery from service outage. A single site database is easily adapted to run in a distributed environment by making use of these facilities.

In the next sections we describe additions that have already been made to Genesis to facilitate distributed data management and then those extensions which still remain

4 Genesis Work Completed

The Genesis project has completed work in two major areas. A redesign and reimplementation of the Locus transaction mechanism, tailored to support distributed data management, is now operational. A record level locking facility is now in use which permits an application program to lock arbitrary ranges of bytes in a file.

4.1 Transactions

It is accepted wisdom that atomic transactions are the correct unit of synchronization and recovery in distributed systems. While several database systems have implemented transactions [Borr81] [Gray79], these implementations have serious limitations which render them unsatisfactory for use as general programming tools. Currently available transaction mechanisms are typically implemented in an application package such as a transaction processor (TP) or a database system. To appreciate the limitations this design imposes, consider an application program X, itself a transaction, that calls an existing database function for part of its work and makes use of a prepackaged transaction processor for another service [Popek83]. Program X also performs some operations directly to the underlying file system. The program may contain the following code fragment:

```
Begin_Transaction      DB
        :
Begin_Transaction      TP
        :
Write(file)
        :
End_Transaction        DB
        :
End_Transaction        TP
```

A machine failure immediately after the first End_Transaction causes the TP transaction to abort while the DB transaction has already committed. This problem results, not from the improper nesting of transactions, but from the lack of coordination between the two mechanisms. Furthermore, in order to back out those file system writes performed directly by X requires building another transaction mechanism on top of the existing ones. Were the transaction mechanism provided in the underlying operating system, it would have been available to all three of these layers.

Also important is the ability to invoke transactions from within transactions [Moss82] [Liskov82]. While not of direct importance to the database system itself, nested transactions are essential to integrating the database into the programming environment. With the ability to nest transactions, programmers are free to compose existing transaction modules just as procedures and functions are composed in programming languages [Mueller83].

An implementation of full nested transactions was operational in the testing environment in UCLA Locus as of 1983 [Mueller83] [Moore82]. This facility allowed an application to create a new process which would run as an atomic transaction. The transaction could itself invoke a module as a subtransaction which would act atomically. That is, when a subtransaction aborted, all of its open files reverted to the state in which they were before that subtransaction was begun. It was possible for a subtransaction to abort (either by choice or because of a failure) while allowing the parent transaction to decide to continue executing.

The previous Locus full nested transaction facility was judged inappropriate for database applications The creation of a new heavy weight process for each transaction is too expensive for database transaction processing Furthermore, the previous implementation did not handle transactions spanning processes at multiple sites.

A new nested transactions facility is now operational in Genesis which solves these problems, and runs with substantially less cost (at the price of a more limited form of nesting). Transactions may still invoke nested subtransactions. However, the failure of any subtransaction causes the entire transaction to abort. There is no synchronization among subtransactions within a single top level transaction; these transactions will share data according to normal locking rules.

Rather than using a system call to cause a load module to be run in a new process as a transaction, Genesis provides the conventional *Begin_Transaction()*, *End_Transaction()*, and *Abort_Transaction()* calls. All code between the Begin_Transaction and End_Transaction, while still part of the same process, behaves as an atomic transaction. Within the top level transaction, there may be many levels of nested subtransactions. Since an abort by any of the subtransactions causes the entire transaction to abort, the nesting of subtransactions has virtually no semantic meaning. It does, however, allow the composition of existing modules which contain, or are themselves, transactions. With this interface, a programmer can surround any critical block of statements with begin/end calls and be assured that the block will be executed atomically. A driving philosophy in our design and implementation is that any code which works correctly, should work identically when surrounded by begin/end transaction calls.

The Genesis transaction mechanism required research in two major areas. First, the fully distributed nature of the actions being synchronized in Genesis make this transaction facility significantly different from previous work. Second, the preexisting Locus transaction mechanism used process boundaries to isolate concurrent transactions. In Genesis transactions, processes are no longer the unit of synchronization and recovery.

4.1.1 Distributed Transactions

As in the first Locus implementation, a transaction may access local and remote data. Genesis also permits a process to spawn subprocesses at remote sites or migrate itself to other sites. Each process must maintain information about the identity of the process which invoked the top level transaction and about all child processes. The operating system provides a process tracking facility which allows us to determine the current location of a process given its process identifier. Parents are able to signal their children through normal Unix interprocess signals and subtransactions are able to communicate their commit/abort status to the top level process.

The operating system maintains a list for each process of files which have been accessed by transactions within that process. When a remote process which is part of a transaction completes, its accessed file list is sent to its parent. The records in each of these files remain locked until the top level transaction commits or aborts. By the time the top level transaction commits, all child processes have completed and the file list for the process which invoked the top level transaction contains all files touched by the whole transaction. The site of the top level process becomes the coordinator for the two-phase commit protocol which will apply the updates to the database.

4.1.2 Transaction Isolation

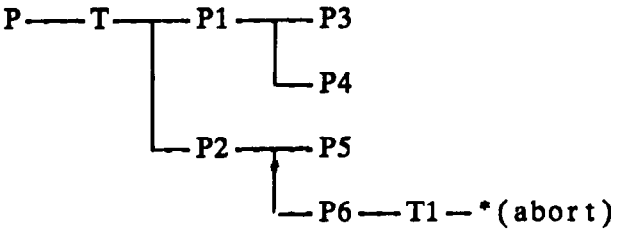
The loss of process boundaries to isolate transactions created several problems. When a process calls Begin_Transaction, it may already have acquired locks and completed updates. If that transaction should abort, the updates made as part of the transaction must be rolled back. However, the updates made before the transaction must remain in effect.

Furthermore, consider the problem of where to transfer flow of control on transaction abort. With a process interface, the system simply rolls back all updates and terminates the process, returning a failure status to the caller. In our implementation, the top level transaction is invoked within a process which may continue to execute even if the transaction aborts. There may be more than one End_Transaction call, so control cannot simply be given to the statement after the

End_Transaction. We are constrained not to solve this problem by changing the C-compiler as its source is not always available and we desire the facility to be usable from any language.

The chosen solution to this problem is to send a software interrupt or signal (say SIGABORT), to the process which initiated the top level transaction If the process has provided code to deal with the signal (i.e. *catches* the signal), that code is run and control is transferred to the desired place. However, if the signal is not caught, it will cause the process to be killed. In this manner, applications which have mechanisms built in to handle an aborted transaction may do so, while those that do not are themselves aborted.

To illustrate this solution in the face of distributed transactions, consider the following scenario: Process P calls Begin_Transaction() to invoke a transaction, T. T then creates processes P1 and P2, each of which also spawn two processes, creating P3, P4, P5, and P6. P6 then invokes a subtransaction T1. Assume that any (or all) of the new processes may have been created on remote machines.



Let us assume P1 completes first. Before going away completely, it must wait for its children P3 and P4. If P3 and P4 terminate successfully, information about any files accessed by them must be communicated up the tree. The operating system, on behalf of P1, adds the list of files accessed by its children to its own and sends that to its parent. The record locks in these files are retained pending the outcome of the entire transaction, T.

Now assume a file being used by T1 becomes unavailable (due to a site failure) causing T1 to abort. The operating system at the site on which P6 is running sends a network message to the site of the top level transaction (say at site 1) informing it that a subtransaction has aborted. Abort messages are sent to the sites of each of P's children within T who, in turn, abort each of their children before dying themselves. Each process rolls back all updated files in its accessed file list by simply discarding the intentions list and shadow pages Finally, the operating system at site 1 issues a SIGABORT to process P informing it that T has aborted. Process P may catch the signal and retry the transaction or give up. If P does not catch the SIGABORT signal, the process is killed.

Inter-transaction concurrency control is achieved by a two-phase locking algorithm enforced by the operating system. All locks are held until the fate of a transaction is determined. Locks are currently on a whole file basis pending integration of the record level locking facility reported in the next section. The transaction mechanism makes use of the shadow page (or intentions list [Lampson79]) scheme already provided by Locus to commit individual files. Multi-file atomic commit is accomplished with the two-phase commit protocol [Gray78] [Lindsay79] and file status logs.

This fully distributed, transparent nested transaction mechanism is operational in Genesis. The new implementation, using only the two phase commit code from the earlier work required about four man months to build and consists of about 4000 lines of C code.

4.2 Record Locking

Synchronization of access to relations, records, or pages of database files is essential to maintaining the consistency of these databases [Gray78]. Genesis record locking provides a transparent, distributed, variable granularity concurrency control mechanism that efficiently fulfills these requirements.

There are significant advantages to providing concurrency controls for data files at the byte granularity level. Database applications that use these record locking mechanisms have the option of controlling access to data items at the page level, record level, or relation level, and are able to dynamically vary the granularity of locking as best suits a particular operation. Also, existing file interfaces in Locus are provided at the byte level of granularity; record locking at this granularity does not unduly complicate the semantics of file synchronization, permits applications to easily define their own file semantics, and allows existing appli-

cations to take advantage of this functionality.

In addition to the difficulty of re-using existing application code, constructing database applications without underlying operating system support can also lead to performance penalties in certain circumstances As reported by Ries and Stonebraker [Ries79], the performance of these database systems may be substantially impacted by locking mechanisms which do not provide controls at a suitable granularity. Database applications which provide their own concurrency controls often provide these controls only at the page or whole database level only, due to the complexity of implementing fine grain concurrency controls. This can lead to performance penalties under certain conditions.

One of the fundamental differences between Genesis record locking and existing record locking mechanisms is its distributed nature. Genesis permits processes located at widely distributed network nodes to synchronize their activities in the same fashion as co-located processes would. This is accomplished in part by using the underlying Locus communication mechanisms to synchronize access at a single network site.

Another unique aspect of the Genesis record locking mechanism is the fact that it is a byte granularity locking mechanism built upon a shadow paging commit mechanism, rather than a logging mechanism. This is achieved by adding a recursive page differencing algorithm to the commit mechanism, as described by Weinstein [Weinstein85]. Although adding this functionality increases the complexity of the basic page-level commit mechanism, the overall complexity of the implementation does not approach that of log-based commit mechanisms.

The locking policy provided by Genesis is identical in functionality to its counterpart in a centralized system. When a file requires synchronization services, the locking mechanism chooses a locking coordinator site (LCS). The LCS is the central location to which locking for that file will be directed. Requests arrive at the LCS from network sites, are processed in FIFO order by the locking policy mechanisms, where access to the requested range of bytes is granted or denied. Replies are passed back to each requesting process. This mechanism is vastly simpler than an algorithm that maintained synchronization in a non-centralized manner.

Another important difference between Genesis and existing distributed locking mechanisms is the granularity provided by the locking mechanisms. Whereas Locus provides file data transport mechanisms on a *page* basis only, Genesis provides byte granularity locking. This is accomplished by separating the locking policy mechanisms and resource coordination mechanisms.

Managing locked records on shared physical pages is handled simply by Genesis. As described above, the Genesis locking mechanisms grant access to ranges of bytes for each file; two requesters may in fact have been granted access to records that are co-resident on a single data page At each site using a data page, the buffer area may contain a copy of the data page. However, only one copy of the data page is considered to be valid at a given time within the network; a copy of a particular page is invalidated as necessary when access rights are relinquished to it. Thus, requesters take turns modifying shared data pages.

Fortunately, it turns out that shared data pages may not substantially affect the performance of a database system. As discussed in [Weinstein85], distributed updates to the same data page tend to occur infrequently, due to the statistical nature of the expected update traffic. In cases where updates occur in sequential records, it will often be the same requester performing those updates.

In order to use the Genesis record locking mechanisms, a typical database application might contain the following code fragment:

```
file = OpenFile(name)
   :
RecordLock(file, range, mode)
```

The *mode* field of the RecordLock request can specify a combination of four orthogonal locking conditions that may be enforced:

XR	Exclude Other Reads
R	Read
XW	Exclude Other Writes
W	Write

Using these modes, a variety of standard locking modes may be specified.

Read/Write	[R,W]
Shared Read/Single Writer	[R,W,XW]
Exclusive Read/Write	[R,XR,W,XW]
Lock for Read/No Writers	[R,XW]

Once a record has been locked, normal read or write operations can be performed on the locked items, following which the changed data items can be *committed*.

```
Read(file, range)
   :
Write(file, range)
   :
Commit(file)
   :
RecordUnlock(file, range)
```

In this example, a set of bytes in *file* are read, modified, the changes are committed, and then the bytes (which were locked earlier) are unlocked and are then available to other processes.

In order to provide support for the Genesis transaction mechanism, locks are internally designated to be in either *normal mode* or *transaction mode*, depending upon whether or not the request originated from a member process of a transaction. In normal mode, locks are acquired and released at the time the corresponding locking requests are made, as would be expected. In transaction mode, however, the two-phase locking protocol is enforced. Unlocking of records is deferred until the end of the current transaction, by marking these locks as *held* by the appropriate transaction. These locks are then unavailable to other transactions until termination of the transaction holding these locks.

The record locking mechanism is now operational in the Genesis distributed operating system The implementation of the record locking mechanism required about 5 man-months of design and coding, and consists of approximately 2500 lines of C code.

5 Ongoing Research

Our ongoing research focuses on building and refining the algorithms and mechanisms required for the support of very large distributed databases, involving potentially hundreds or thousands of computers, in local and wide-area networks.

There are three primary areas of interest. The first of these is the development of data storage replication mechanisms, which will provide the means to improve the reliability, availability and performance of distributed databases supported by Genesis. The second area of investigation concerns the applicability of the Genesis architecture to low bandwidth, wide-area networks. Lastly, ongoing work examines the problems of managing thousands of nodes in a single network.

Replication of data files provides an effective means of improving both performance and reliability of distributed databases. The Locus operating system currently supports distributed file replication and automatically maintains the consistency of multiple copies of data. Locus allows updates to replicated data during partitioned operation using a primary site algorithm.

A more general strategy for partitioned operation is desirable in distributed database systems. In our view, it is essential that updates be permitted to data even when some copies of that data are unavailable. [Parker81] shows how inconsistencies resulting from independent updates during partitioned operating may be detected. Some work has already been done by [Faisol81] and [Guy85] on approaches to automatic recovery of inconsistent data.

A distributed database system can take advantage of the distributed transparent name structure of Genesis, to provide support for naming (catalog) mechanisms [Thiel 83]. Name service is the generalization, in a distributed environment, of directories in a single machine operating system, or of analogous functions in a database schema; specifically, the mapping of names to the location of the named entity In a distributed database, the problem is potentially substantially increased. Name mapping tables must be at least partially replicated, and those copies kept mutually consistent. However, unlike simple operating system directory sys-

tems, the database name-to-location mapping may involve functions of the database values themselves; i.e. storing all *Los Angeles* tuples in a *Payroll* relation on the southern California computer center of the company. As described by Thiel [Thiel 83], relations may be named as files, and their names appear in the database catalogs. In the Genesis system, the underlying operating system file replication facility is being used to store extended database schema information.

Research is also being pursued toward the development of resource controls that permit dynamic relocation of replicated file and directories, which can improve data locality, and hence lead to greatly improved performance.

A second area being investigated is support for low bandwidth, wide-area networks. The approach outlined for local networks may be extended to include the wide area network environment, where available communication bandwidths are much more limited, and network delays are far greater. In local networks, there is only a slight penalty for accessing a remote resource. The very short delays and high bandwidths present in local networks provide access to remote resources (such as remotely mounted files) at rates and delays comparable to those available for access to local resources. Simple operating system level algorithms, such as paging remote directories for local interrogation, or accessing data files, can provide acceptable performance in these circumstances. Wide area network delays of several tenths of seconds make such a strategy unacceptable.

Substantial changes have been made within Genesis, at the underlying communications protocol level, and at the internal operating system level to address these issues. More conservative paging algorithms, protocols with higher semantic content, and other techniques are being applied to solve this problem. Sheltzer [Sheltzer85] discusses these areas in detail.

The third area of research now in progress is concerned with the critical tradeoffs required to support substantial numbers of nodes (i.e. >10,000) in a single network, at both the operating system and application programming level. This research is attempting to determine how much internode communication and how much state information is necessary and sufficient to ensure the proper operation of large scale networks.

It has been shown that, in a large scale distributed database system, the underlying architecture must not need volatile global agreements, nor may it require each site to maintain a consistent view of the state of other members in the network; this may impose unacceptable communication or computing costs. Genesis generally takes a "bilateral view": sites maintain detailed synchronized state with respect only to those other sites with which they are actively interacting.

Another issue of prime concern in large networks is heterogeneity of computing resources. Large distributed database systems can expect to be hosted by a network with a number of different machine types, whose instruction sets as well as data representations do not match. A general facility for support of strong data types in the operating system would be attractive, but is not considered essential. Besides the mechanisms needed for the operating system itself [Locus84], little more than information present in the database schemas, plus additional data access routines, is necessary in Genesis.

Solutions in these research areas are important to the long-term viability of the approach to database support espoused by Genesis.

6 Conclusions

Experience with the ongoing development of Genesis makes it clear that, if a distributed system base is available, the architecture described in this paper provides substantial advantages over conventional approaches to distributed data management. Development costs may be greatly reduced, and the resulting system is more usable, with little penalty in flexibility or performance. The development experience described here in the case of Ingres is a dramatic example of how inexpensive distributed database functionality can be, both in terms of development effort and elapsed time.

By eliminating the barriers imposed by machine boundaries, both at the operating system and database levels, it is far easier to achieve the potential which distributed systems possess. The one clear limitation in the Genesis design is that all sites in the network must run the Genesis software;

operating system and database system heterogeneity is not directly addressed. While there are approaches to this problem which are natural extensions of the design architecture discussed in this paper, these extensions are beyond the scope here.

Though parts of Genesis are still under development, enough has been built and used that we have confidence in the conclusions drawn throughout this discussion.

References

[Borr 81] A.J. Borr, "Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing," Proceedings of Very Large Database Conference, 1981.

[Epstein 78] R. Epstein, M. Stonebraker, E. Wong, "Distributed Query Processing in a Relational Data Base System," Proceedings 1978 ACM-SIGMOD Conference on Management of Data, Austin, Texas, May 1978.

[Gray 81] J. Gray, "The Transaction Concept: Virtues and Limitations," *Proceedings of the Seventh International Conference on Very Large Data Bases*, Cannes, France, September 9-11, 1981, pp. 144-154.

[Gray 78] J. Gray, "Notes on Data Base Operating Systems," Operating Systems An Advanced Course, *Lecture Notes in Computer Science 60*, Springer-Verlag, 1978, pp. 393-481.

[Guy 85] R. Guy, "File Replication in LOCUS: A Distributed Operating System," Masters thesis, Department of Computer Science, University of California, Los Angeles, 1985.

[Haas 82] L.M. Haas, P.G. Selinger, E. Bertino, D. Daniels, B. Lindsey, G. Lohman, Y. Masunaga, C. Mohan, P. Ng, P. Wilms, R. Yost, "R*: A Research Project on Distributed Relational DBMS," IBM Research Report RJ 3653, IBM Research Laboratory, San Jose, CA., October 21 1982.

[Held 75] G.D. Held, M.R. Stonebraker, E. Wong, "INGRES - A Relational Data Base System," Proceedings of the National Computer Conference, Vol. 44, 1975.

[Lampson 79] B.W. Lampson and H.E. Sturgis, "Crash Recovery in a Distributed Data Storage System," XEROX Palo Alto Research Center, April 1979.

[Lindsay 79] B.G. Lindsay, P. Selinger, C. Galtieri, J. Gray, R. Lorie, T. Price, F. Putzolu, I. Traiger, and B. Wade, "*Notes on Distributed Databases*," IBM Research Report RJ2571(33471), IBM Research Laboratory, San Jose, CA, July 14, 1979, pp. 44-50.

[Liskov 82] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," Proceedings of the Ninth Annual Symposium on Principles of Programming Languages, Albuquerque, NM, 1982.

[Locus 84] "*The Locus Distributed System Architecture*", Edition 3.1, Locus Computing Corporation Technical report, June 1984.

[Moore 82] J.D. Moore, "Simple Nested Transactions in LOCUS: A Distributed Operating System," Master's Thesis, Computer Science Department, University of California, Los Angeles, 1982.

[Moss 82] J. Eliot B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, M.I.T., 1981.

[Mueller 83] E. Mueller, J. Moore, G. Popek, "A Nested Transaction Mechanism for LOCUS," *Proceedings of the Ninth Symposium on Operating Systems Principles*, Bretton Woods, NH, October 10-13, 1983.

[Page 85] T.W. Page, Jr., "Distributed Data Management in Local Area Networks," *Proceedings of ACM SIGMOD/SIGACT Conference on Principles of Database Systems*, Portland, Oregon, March 1985.

[Parker 81] D.S. Parker, Jr., G.J Popek, G. Rudisin, A. Stoughton, B J Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983, pp. 240-246.

[Popek 83] G. Popek, G. Theil, "Distributed Data Base Management System Issues in the LOCUS System," IEEE Database Quarterly, June 1983.

[Popek 81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G Rudisin, and G. Theil, "LOCUS: A Network Transparent, High Reliability Distributed System," Proceedings of the Eighth Symposium of Operating Systems Prinicples, Pacific Grove, CA, December 1981.

[Ries 79] D.R. Ries, M.R. Stonebraker, "Locking Granularity Revisited," ACM Trans. on Database Systems, 4, June 1979, pp. 210-227.

[Rothnie 80] J.B. Rothnie, P.A. Bernstein, S.A. Fox, N. Goodman, M M. Hammer, T.A. Landers, C L. Reeve, D.W. Shipman, and E. Wong, "Introduction to a System for Distributed Databases (SDD-1)," ACM Trans. on Database Systems, 5,1 March 1980, pp. 1-17.

[Sheltzer 85] A.B. Sheltzer, "Network Transparency in an Internetwork Environment," Ph.D. Dissertation, Department of Computer Science, University of California, Los Angeles, 1985.

[Stonebraker 76] M.R. Stonebraker et. al., "The Design and Implementation of INGRES," TODS 2,3, September 1976.

[Stonebraker 77] M.R. Stonebraker and E. Neuhold, "A Distributed Database Version of Ingres," 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977.

[Stonebraker 81] M.R. Stonebraker, "Operating System Support for Database Management," Communications of the ACM, Vol. 24, No. 7, July 1981, pp. 412-418.

[Stonebraker 83] M.R. Stonebraker, J. Woodfill, J. Ranstrom, J. Kalash, K. Arnold, and E. Anderson, "Performance Analysis of Distributed Data Base Systems," University of California, Berkeley Memorandum No. UCB/ERL 83/55, 25 July 1983.

[Thiel 83] G. Thiel, "Partitioned Operation and Distributed Data Base Management System Catalogs," Ph.D. Dissertation, University of California, Los Angeles, 1983.

[Walker 83a] B. Walker, "Issues of Network Transparency and File Replication in the Distributed Filesystem Component of LOCUS," Ph.D. Dissertation, University of California, Los Angeles, 1983.

[Walker 83b] B. Walker, G. Popek, R. English, C. Kline, and G. Theil, "The LOCUS Distributed Operating System," *Proceedings of the Ninth Symposium on Operating Systems Principles*, Bretton Woods, NH, October 10-13, 1983.

[Weinstein 85] M.J. Weinstein, "Some Performance Aspects of Shadow Paging Mechanisms," Locus Memorandum #21, Department of Computer Science, University of California, Los Angeles, 1985.