1986

The Design of an Adaptable Distributed System

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

John Riedl

Report Number:
86-580

THE DESIGN OF AN ADAPTABLE DISTRIBUTED SYSTEM

Bharat Bhargava
John Riedl

CSD-TR-580
February 1986

The Design of an Adaptable Distributed System †

Bharat Bhargava
John Riedl

Computer Sciences Department
Purdue University
West Lafayette, In 47906
(317) 494-6013

ABSTRACT

There is a need to design large database systems that are not rigid in their choice of algorithms and are responsive to faults/failures and performance degradation. To attack this challenge, we formalize and experiment with design principles that allow the implementation of an adaptable distributed system. By adaptable, we imply that systems can be reconfigured at run-time based on performance and continuity of operations requirements and load conditions. Our research focus is on algorithms for concurrency control, resiliency to site failures, network partitioning, and failure of communication systems. The strategies for dynamic reconfiguration of the software algorithms and determining their impact are being studied both theoretically and via experiments on a prototype system called **RAID** being developed at Purdue. We describe a layered design for a distributed operating system with distributed protocols that can be modified -- or even completely changed -- while the system is running. This capability will be a help in tuning the system to improve its performance and reliability. In addition, the increased flexibility of this design makes it suitable for diverse applications, and capable of incorporating new distributed systems technology as it becomes available, unlike existing systems.

1. Introduction

Current distributed systems provide a rigid choice of algorithms for software implementation. The design decisions are based on criteria such as computational complexity, simulations under certain assumptions, or at best limited empirical evidence. The desired life cycle of a system is at least several years. During such time new applications surface and distributed systems technology advances, making the earlier design choices less valid. In addition during a small period of time (within a 24 hour period) a variety of load mixes, response time requirements and reliability requirements are encountered. Different concurrency control and recovery algorithms are suitable for different load, performance, and reliability requirements [Bhar 84]. Only an adaptable distributed system can

† This research is supported in part by the U.S. Department of Transportation and Sperry Corporation.

meet the various application needs in the short-term, and take advantage of advances in technology over the years. Such a system will adapt to its environment during execution, and eliminate the overhead of redesigning the complete system because of outdated design choices.

For instance, distributed systems which provide some method of handling the network partition problem [Davi 84], use a method based on one of two major approaches. The conservative approach relies on careful protocols that ensure that all changes that happen in any partition maintain consistency across all partitions [Mino 82]. In contrast, the optimistic method permits inconsistency during the partition, and includes a merge phase during which all of the sites in the complete network again agree on a consistent database state [Davi 84]. In systems which have heavy access to shared data items the conservative approach is based on the reasonable assumption that the work required to merge partitions that have become inconsistent would be greater than the savings achieved through the higher concurrency possible during the partition. On the other hand in many systems it is reasonable to assume that most of the data items in different partitions can be safely accessed simultaneously. This is true, for instance, for traditional time-shared file systems such as UNIX. Then when the partitions are merged, any activity which did violate the consistency requirements of the system as a whole can be rolled back or compensated [Bhar 82b]. Different applications require different solutions to this problem, but few current systems offer more than one alternative.

Another example is concurrency control. The degree of concurrency provided by a system is affected by the algorithm which is implemented. Unfortunately, these algorithms use different data structures and even have different information requirements. So a system built with one concurrency control approach is likely to require significant effort to convert to a different method. We consider this example again in section 4.2.2.

The selection of the basic protocols is made early in the design phase of a system. Unfortunately, this selection determines the applications for which a new system is suitable long before that system is ready for applications. This severely restricts the usefulness of the completed system.

Our solution to this problem has two major thrusts. First, each of the major subsystems will be parameterized to permit tuning or even replacement while the system is running. For example, the partition control protocol in our design uses a conservative approach when rollbacks would be expensive, and an optimistic approach when the probability of cyclic conflicts is low. Second, we work to restrict the possible communication between the protocols by designing our system in layers with clearly defined communication paths between the layers. This allows new protocols to fit into our system without disturbing protocols on other levels.

In this paper we discuss the ideas of layering, algorithm replacement, and real-time adaptation. Section 2.1 describes our layering scheme, and section 2.2 contains several examples of real-time adaptation. Section 3 discusses ways in which the ability to

change algorithms in response to circumstances can be used to improve the performance and reliability of a distributed system. Section 4 describes the RAID experimental distributed system which we are using to test our ideas. Finally, section 5 suggests future work towards building adaptable distributed systems and section 6 is the conclusion.

2. Adaptability

We identify two main principles for our research. First, we view the system as consisting of layers with clearly defined communication paths between the layers. This allows introduction of new protocols at different levels without affecting the subsystems at other levels. The second principle involves switching from one class of algorithm to another class. We research both of these ideas, and consider several examples of their application in this section.

2.1. Layering

In this section we describe our preliminary layering scheme. This discussion is meant to provide a framework within which to discuss the rest of the system. To keep our design flexible we suggest a class of layering schemes that support the functionality needed for our system. We will concentrate on a particular layering within this class that is especially suitable for adaptability. However, any layering scheme from the class would be acceptable. Our scheme is quite similar to existing layered operating systems [Tane 72] except that we de-emphasize traditionally important areas such as the I/O and memory management sub-systems in favor of the support which we offer for distributed systems.

Our layering scheme is depicted in figures 1, 2, and 3. Figure 1 provides abbreviations which are used in the diagrams. Figure 2 is a dependency graph which shows the interaction between the layers. The fundamental requirement for a layered system is that inner layers must not depend on features provided by outer layers. Our dependency graph encodes this relationship with directed dependency edges. For instance, the directed edge from the RPC layer to the CO layer means that our remote procedure call mechanism will use primitives from the basic communications layer, and must be above it in the eventual layering scheme. This dependency graph depicts the rules that determine the class of layering schemes that are most accommodating to our ideas. The dependencies involving the LIO, LMM, and UMM layers are standard to layered operating systems. Our upper level I/O system (UIO) is responsible for logging I/O requests from transactions to provide undo/redo capabilities for atomicity and reliability. Thus it must be below the transaction level (TR) and above the stable storage level (SS). Remote procedure call (RPC) clearly must use the communications system (CO). TR must also use the CO, and will need UIO and SS in addition. Finally, the partition control layer (PC)

requires CO, and provides the error-free virtual network on which TR is built.

Figure 3 is a possible layering based on these dependencies. This layout has many characteristics that we feel are important in the design of an adaptable distributed system, but we stress that it is only one of several possible. In the next few subsections we explain the choices that we have made in constructing this diagram. In cases in which other alternatives seem almost as attractive as the one which we chose, we briefly defend our choice and suggest situations in which the alternatives would be preferred. In section 4.2.1 we relate these layering ideas to the RAID distributed system.

2.1.1. Transactions

The transaction concept has proven itself as the correct model for reliable distributed processes [Gray 79], so we concentrate on providing support for distributed nested transactions at the lowest possible level. From the second diagram it can be seen that remote procedure calls, for instance, can be executed within transactions, and are subject to the rules of commit and abort. Since transactions must be reliable through site and network failures, the protocols for coping with site failure and partition are below the transaction level. It should be noted that these failures will be handled transparently with respect to outer layers. Thus, the transaction level can be written to run on a errorless virtual network of failure-free sites.

2.1.2. Communication

The communications layers provide services ranging from a basic message passing facility on which the distributed system is built to a remote procedure call mechanism. The most important part of this system is a reliable datagram service, which guarantees that either the datagram is sent correctly or that an error indication is returned. The choice of datagrams as the basic unit of communication rather than a higher level service such as virtual circuit [Tane81] is based on the fact that most system level communication in a distributed system consists of discrete packets of data. The advantages of higher-level communication are not needed for this sort of communication.

However, users will often wish for higher level services, so we provide a reliable remote procedure call (RPC) mechanism for most user-level interprocess communication [Shri 82]. RPC has become an accepted mechanism for IPC, and has several major advantages as a method for building a reliable system. Most notably, the semantics of rolling back a datagram message that has been sent or received are difficult to define and hard to implement. On the other hand, the atomicity requirements for RPC can be provided easily by a sub-transaction of the transaction attempting the RPC [Lisk 83].

Finally, a broadcast capability is provided to make distributed commitment protocols easier and more efficient. The broadcast routines use hardware broadcast if it is

provided by the network. Otherwise, a system-level simulation is still more efficient than a user-level simulation.

This three part approach satisfies the necessity for efficiency in system communication, without sacrificing convenience for the programmer. The system designer has both point-to-point and broadcast datagrams while the user-level programmer can make use of the RPC abstraction.

2.1.3. Input/Output Systems

The three major I/O systems are the lower I/O system, the upper I/O system, and the stable storage system. The lower level I/O is the support needed by the virtual memory management system. For the most part, this consists of simple calls for reading and writing raw data. The stable storage system provides I/O facilities suitable for use as a log [Lamp 78]. Because a log is intended to improve the reliability of the system as a whole, it is essential that the log itself be more reliable than the rest of the system. This level is intended to be hidden from users of the system, but is used by upper levels to provide recoverable I/O. Finally, the upper level I/O routines provide the I/O interface seen by applications programmers. In particular, this level offers a reliable read/write protocol permitting replication and providing location independence [ref-replicated]. Section 2.3.2 offers more detail on the upper level I/O protocol.

2.1.4. Memory Management

Memory management is not a primary focus of our current work. Certainly the work that is being done in the area of distributed memory management [LLHS 85] is important, but in order to make our research applicable to current distributed systems we have chosen to use more standard methods. In particular, our memory management software consists of separate units at each site which have no interaction with one another. As shown in the diagrams, we provide for virtual memory management since that is an important part of modern systems. However, memory management is safely below the network level, and is presumed to be implemented in a standard manner [Denn 70].

2.2. Algorithm Replacement

2.2.1. Concurrency Control

The concurrency control system is one of the most important candidates for parameterization. The hierarchy among the classes of algorithms for distributed concurrency control is shown in Figure 4 and was developed in [Hua 82]. Each of the rectangles represents a set of histories that is accepted by a particular class of concurrency control algorithms. The containment relationship between different classes induces a partial order of concurrency control classes. At the top is the NP-complete class SR that contains transaction histories that preserves database consistency. At the bottom is the trivial class that allows only serial histories, but can be decided in constant time. Practical algorithms such as two-phase locking and those based on timestamp mechanisms fall somewhere between these extremes. Here the tradeoff is between the efficiency of the algorithm and the degree of concurrency that it permits. In section 3.1 we list some of the parameters that determine which concurrency control algorithm is best suited for an application.

2.2.1.1. Concurrency Control Adaptability

The goal of changing concurrency control methods while the system is running can be easily achieved. Simply stop entering new transactions into the system, wait until all transactions are completed, and start allowing transactions to enter the system again. This solution has two flaws that make it unacceptable. First of all, the concurrency of the system during the conversion will be dramatically lowered. Second, the conversion cannot begin until all transactions that were running when the conversion decision was made have completed. In the presence of long transactions, this delay will be unacceptable.

For these reasons, we are working on ways in which concurrency controllers can be switched without first stopping all transaction processing. The primary difficulty with this approach is that different concurrency control methods use quite different data structures. We are working on two solutions to this problem. The most obvious is to try to find a general data structure that permits efficient concurrency control for all methods. In section 4.2.2 we describe a data structure that supports all locking and timestamp concurrency control methods without modification. The second approach is to find efficient ways to convert between the data structures needed for the various methods efficiently. For instance, we are looking at ways to convert a lock table to approximate timestamp information quickly. Along with either of these approaches we need a protocol to allow limited transaction processing while the concurrency control method is being changed.

2.2.2. Partition Control

When a communication system failure occurs, two or more sets of operational sites may partition and find themselves unable to send updates to replicated copies. Since

availability of the latest values of the database items and the continuation of the transaction processing cannot be prohibited, a variety of protocols [Bhar84, Davi84, Bhar86] have been suggested.

As an example, the token-based scheme [Mino82] can be used to identify the copy of each database item that can be accessed during a network partition. The tokens can be stationary (assigned to a unique copy) or be moved from one copy to another based on the system's requirements. There is a certain amount of overhead in managing the tokens that can be lost in the worst case.

Another protocol for dealing with network partitions is to allow the database items in the partition with the majority of sites to be accessed by transactions. This protocol blocks all transactions in the partitions with the minority of sites and in the worst case may block all transactions if no partitions has a majority.

As a third alternative, all partitions can process optimistically (hoping few rollbacks) all transactions but only semi-commit (transactions can be rolled back) the results and ensure consistency when partitions merge [Bhar82b].

Either of these alternatives are appropriate under different conditions. If the duration of network partition is small and the probability of cyclic conflicts is low, the third alternative can perform very well and increase throughput and response time. In addition, it is resilient to multiple failure, of partitions themselves. One can also switch from this protocol to the protocols based on majority of sites or the token approach. They may be necessary for some real-time constraints or priorities of sites or certain database items.

Another dimension of flexibility arises when the network partitions merge. In the token approach, no additional work is required. In the majority site approach, the sites blocked out from any processing need to be integrated [Bhar86] with active sites. In the optimistic approach, the transaction processing in different partitions need to be serialized. This can be done by merging various dynamic conflict graphs (DCG) [Bhar82] and maintaining acyclic property. This task itself can be done in several ways. Two of them are listed as follows:

a) Combine all DCG's and check for acyclicity. This idea is too optimistic. The rollback of transaction may always be needed and to minimize the rollback in NP-complete [Davi84].

b) Assign a weight to each transaction in each DCG. Consider the DCGm with maximum weight. Immediately commit all transactions in DCGm. Select transactions from other DCG's that do not create cycle with DCGm and commit them. Abort the cycle producing transaction.

Once again the choice depends on the reliability, performance, and other requirements of the system. But the system design should permit adaptability to any variation of protocols during partitions and when the merging of partitions takes place. This can be achieved if comprehensive data structures are used for the bookkeeping during regular

processing and certain conditions are maintained when the switching from one class of protocols to another occurs.

2.2.3. Read/Write Voting

Distributed systems with replicated data must use a protocol that ensures that updates are observed consistently by all transactions. Many systems use one of the read/write voting protocols categorized by Gifford [Giff 79]. Essentially, each of these protocols chooses some number α, and requires that a successful read must read at least α sites, and a successful write must write at least $n-\alpha$ sites, where n is the total number of sites in the system. These protocols range from the popular methods like read one/write all to special weighted voting methods for unusual situations. Many variations of these protocols have been employed in actual systems. However, almost any single choice is easy to criticize for some applications. Therefore, our upper level I/O system, the read/write system, will be parameterized to allow it to change dynamically among various of these protocols *while the system is running*. During periods with many more reads than writes such as overnight system consistency verification the system could use read one/write all, but during periods with more writes than reads such as restoring the database to an earlier state from a backup tape the system could use read all/write one. Of course, most of the time the mix of reads and writes would dictate a compromise between the two methods. Selecting the correct protocol is a complex task.

There are many other examples of ways in which a distributed system could be parameterized. Good candidates include the choice between centralized and decentralized distributed commitment protocols and the site recovery algorithm. Choosing the correct parameters and fitting them into the system is an important question.

3. Real-Time Adaptation

One of the goals in designing a system is to be able to adapt to changing circumstances as they occur. In the previous section we discussed ways in which the pieces of such a distributed system could be changed. Now we consider the problem of deciding which combination of the available choices would be best for the operation of the system.

3.1. Parameters

In order to choose the proper components at a given time certain information about the activities in the system must be gathered. Each of the parts of our distributed system will be responsible for gathering information about its operation and supplying that information to a monitor process that will make these critical decisions. The following parameters are among those that are known to be important [Bhar 84].

multiprogramming level: the number of transactions active at any given time

arrival rate: the speed at which transactions are queuing to enter the system

response time: the average real time to completion of a transaction

CC overhead: the average amount of time that a concurrency control method takes to make the commit/abort decision for one transaction

rollback cost: the cost of rolling back a transaction that must be aborted versus the cost of blocking transactions from running

transaction conflict: a measure of the amount that transactions compete for access to database items

update/read-only: the ratio of transactions that make database updates to those that only read

reads/writes: the ratio of the average number of reads made by a transaction to the average number of writes

transaction size: the average computing resources required for a single transaction

deadline: are there some transactions that have real-time constraints?

semantics: are there special application-specific semantics that affect the concurrency control decision?

The relationship between individual members of this group of parameters and individual concurrency control algorithms is documented [Bhar 82]. However, the interactions between various parameters is complex and difficult to analyze. Providing the capability for a system operator to tune the system based on a complete set of performance parameters is certainly an advantage over current systems, but some form of automatic adaptation would be desirable.

3.2. Expert System

Unfortunately, the relationships between these parameters are difficult to model precisely. Experts who have researched the problem have a good understanding of the factors influencing these decisions, but are not able to provide analytic models to solve the problem. An expert system would be a good tool for controlling the adaptation strategy. This expert system would maintain a knowledge base consisting of a group of parameters affecting adaptation, with deduction rules based on the known relationships between the parameters. It will receive data from each of the components of the system periodically, make decisions about the preferred state of the system, and communicate those decisions to the components that should reconfigure or switch protocols. The knowledge-base will grow based on the past experience of the expert system.

The statement of this problem is easy enough, but it has some hidden complexity. For instance, one of the parameters which such a system would have to modify is the frequency of data collection and the amount of information that is collected to make the

decision! During periods of gradual change, hysteresis should be high which indicates that infrequent examination of the status would be enough. On the other hand, when changes are occurring rapidly it might be worth the overhead of collecting the data more frequently to ensure that a good protocol is in use at all times. It is a demonstration of Heisenberg's Principle that the closer we monitor the system the more we impact its performance.

Figure 5 is a schematic diagram showing the components of our expert system and its relationship to the rest of the system. The goals for this system are developing an appropriate set of parameters for monitoring system performance, and creating an set of inference rules that contains the important relationships between these parameters and the various algorithms.

4. Experimental Effort

RAID is an experimental system being developed on VAXen running the UNIX operating systems in order to investigate the principles necessary to build a high performance, reliable, and adaptable distributed database system. Since a variety of system control functions are needed to increase the integrity, concurrency, and reliability of a distributed database system, this effort focuses on the experimental study of the principles that allow the dynamic selection of these algorithms and the reconfiguration of the system.

4.1. Experimental Prototype - RAID

Currently there are six major subsystems in RAID: Parser (PAR), Access Manager (AM), Action Driver (ACT), Auditor (LOG/DIFF), Atomicity Controller (AC), Concurrency Controller (CC). The relationships between these subsystems is depicted in figure 6. PAR accepts transactions expressed in a relational calculus (INGRES-QUEL type) language and produces read/write actions. These actions are processed by ACT which communicates with AM for I/O and AC for commitment of transactions across the distributed system. AC validates transactions for local serializability with CC. Before posting the updates in the database, ACT goes through the auditor that can use either a log or a differential-file based system. All sites in the system contain all six subsystems and can process local transactions independently and global transactions via the communication system that ties all the ACs together. This communication system is based on the datagram service provided by the network level.

4.2. Status and Project Plans

Currently the system provides two choices for the auditor/back up system and six choices for concurrency controller. The switching from one choice to another is done statically. Our major goals are to permit the dynamic re-configuration described in this

paper, and to implement alternate choices for the other sub-systems. Effort is under way to implement a system to deal with site failures and network partitions.

4.2.1. RAID Layering

The RAID system existed before our layering ideas, so in many respects it will have to be changed to fit them. RAID will be a tool for testing various layering schemes for ease of implementation and functionality. To date we use the memory management and lower level I/O (LMM, LIO, and UMM) of the UNIX system [Thom 78]. We have implemented an easy to use datagram service on top of TCP/IP UDP datagrams [Post 81]. Transactions (TR) run on top of this layer, using only the upper level I/O, upper level memory management, and communication services of the lower levels. This part of the system fits the layering scheme of figure 3 perfectly.

We have not implemented the partition control (PC), stable storage (SS), or remote procedure call (RPC) layers. Partition control must be implemented transparently with respect to the transaction layer. The transaction layer currently stores its log files on ordinary disk files. When the stable storage system is complete it should be used for log files. Finally, remote procedure call will be built on top of the transaction level, providing users with reliable inter-process communication.

4.2.2. Concurrency Control Implementation

Our first attempt at designing an adaptable concurrency controller has been to design a data structure that supports any locking or time-stamp based concurrency control method. This seems like a hard problem, especially with the need for efficiency. Our solution starts by simplifying the problem. Recently, researchers have discovered a special class of **optimistic** concurrency control algorithms that keep enough state information about each completed transaction to decide whether it can commit [Bhar 82]. Then the actual decision can be delayed until a more appropriate time, allowing for improved concurrency. The implementation of these optimistic protocols has suggested a new method, which we call **validation** for implementing the usual conservative concurrency control algorithms. Essentially, the idea is to permit all transactions to run to completion without enforcing any restriction on database access. Then as the transactions complete, their history is examined to see if it satisfies a particular concurrency control method. Thus, the validation approach to two-phase locking is to determine after a transaction has run whether there is any assignment of locks that would satisfy the normal two-phase locking requirement and still allow the transaction to run as it did. If there is such an assignment, the transaction is committed; otherwise it must be aborted.

The data structure that we use is shown in figure 7. The transaction history consists of transactions T1 through T4. Each transaction has a number of atomic read or write actions, each of which accesses one database item. These transactions have already been committed but must still be kept in the history because some transaction that was running

concurrently with them has not yet completed. They will be removed from the history when all the transactions that were running concurrently with them have completed. When transaction T5 completes it must be validated with this history. Figure 8 contains the validation routine for simple locking in which a read or write locks an item until the transaction completes. We also use as an example the timestamp validation routine in which a transaction commits only if every item that it updates does not change between the read and write actions. T5 will be committed under either of these algorithms. The history including T1 through T4 and T6 is not serializable, and must be aborted by any correct concurrency control method. T7 will also be aborted by the simple locking algorithm, but can be committed under time-stamping.

The advantage of using this approach is that it keeps enough information about executing transactions to allow a variety of concurrency control methods to be applied. When other events suggest that a new concurrency control method should be used, the current one can be replaced without converting any data structures. This allows both rapid conversion and efficient execution.

The correctness of most concurrency control methods depends on having all transactions that run concurrently follow the same method. This suggests that we must stop entering new transactions into the system until all the currently running transactions are completed, apply the validation method to all of them, and then start up the system again with a new concurrency control method. This approach is shown in figure 9, where method A is being changed to method B. If the concurrency control methods A and B have no overlap, this is the best that can be done. However, as shown in figure 4, many of the methods overlap substantially. In the example of figure 10, in order to convert from A to B we need only force the concurrently executing transactions to be acceptable to B, rather than to be completely halted. Thus, if we can force the current history to be in the intersection of A and B, it is safe to switch methods. Fortunately, with the validation method this is easy to do. For each transaction that completes, we run the validation routines for both A and B, and only allow the transaction to commit if both routines permit commitment. Then after all of the transactions that were started while method A was in place are out of the system, method B may be run alone.

4.2.3. Parameterization

As described in the previous section it is possible to parameterize the concurrency control system. In the near future, we plan to parameterize the following subsystems and experiment with them:

atomicity controller: We will implement several commit/termination protocols.

partition control: At least two different site failure and network partitioning algorithms will be implemented.

I/O system: The I/O system will be parameterized using the scheme of section 2.1.3.

We are also studying the principles of switching from one algorithm to another for different subsystems at run-time. The switching criteria will be based on a variety of parameters such as multiprogramming level, degree of conflict, mix of transaction size and type (read-only versus update), available semantics and real-time requirements, workload (arrival rate, overheads, response time) and so on. We are beginning the design of an expert system that will automatically reconfigure the system to changing environmental conditions while it is running.

5. Further Work

An important area for further research is to develop ways in which other distributed systems algorithms can be parameterized. Along with this research, more information will be needed about the effects of various parameters on the algorithms. Of particular importance are performance results which describe the effect of changes in one subsystem on other subsystems. For instance, the choice of a concurrency controller is almost certain to impact the choice of a commit or partition control protocol. Certain algorithms for different subsystems cooperate efficiently. as an example, the optimistic method for concurrency control is compatible with a similar protocol for the network partition problem.

In fact, our goal of designing an expert system to help manage the choice of algorithms based on various information about the system meshes well with this research need. Such an expert system would be in a good position to record observations about the effects that its choices made on later system performance. We hope to have a meta-expert-system that modified its behavior based on data collected in this way.

This work may influence future designers of distributed algorithms to generalize their algorithms so that they can be used to maximum benefit in an adaptable system. For instance, the multi-dimensional timestamp concurrency control method [Leu 86] can be parameterized to permit different types of concurrency.

6. Conclusion

Distributed systems contribute towards building highly reliable and available systems. Unfortunately, the state of the art of building a distributed system is to select an ad hoc set of protocols based on a particular application, and then to build the system from the ground up. This is very wasteful, especially in view of the fact that many distributed algorithms are closely related, sharing the same data structures and similar subroutines. Further, this approach tends to decrease the reliability that makes distributed systems so attractive in the first place.

Our distributed system design has pieces which are interchangeable to a large degree. In particular, we have specified the relationships between pieces of our design in a layered fashion, so that new or different algorithms can be integrated with minimum

effort. In addition, we are designing the protocols that are used in the system so that they can be modified or tuned while the system is running. This tuning can be done by a well-trained operator through a user interface, or automatically by an expert system front-end.

This type of design has not been possible until recently, because not enough was known about the form that distributed protocols would take and performance data were not available. But now, with many concrete ideas, we feel that a careful design can allow for the desirable flexibility.

References

[Bhar 82] Bhargava, B., "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking", *Proceedings of the Third IEEE CS DCS Conference*, pp 508-517, Oct 1982.

[Bhar 82b] Bhargava, B., "Evaluation of Network Merging based on Optimistic Concurrency Control and Rollback Semantics", *IEEE International Conference on Computer Software and Applications (COMSAC-82)*, Nov, 1982

[Bhar 84] Bhargava, B., "Performance Evaluation of Reliability Control Algorithms for Distributed Database Systems", *Journal of Systems and Software*, Vol. 3 pp 239-264, July, 1984

[Bhar 86] Bhargava, B., "Concurrency and Reliability in Distributed Systems", Van Nostrand and Reinhold, 1986 (To appear), edited book.

[Davi 84] Davidson, S. "Optimism and Consistency in Partitioned Distributed Database Systems", *ACM Transactions on Database Systems*, Vol 9., No. 4, pp 456-482, Sept 1984.

[Denn 70] Denning, P., "Virtual Memory", *ACM Computing Surveys*, vol. 2, no. 3, pp 153-189, Sept 1970.

[Giff 79] Gifford, D., " Weighted Voting for Replicated Data", *Proceedings of the Seventh ACM Symposium on Operating System Principles*, pp 150-161, Dec 1979.

[Gray 79] Gray, J., "A Transaction Model", *IBM Research Report*, 1979.

[Hua 82] Hua, C. and B. Bhargava, "Classes of Serializable Histories and Synchronization Algorithms in Distributed Database Systems", *Proceedings of the Third IEEE CS DCS Conference*, pp 438-446, Oct 1982.

[Thom 78] Thompson, K., "UNIX Implementation", *The Bell System Technical Journal*, Jul-Aug 1978, 57(6).

[LLHS 85] Leach, P., P. Levine, J. Hamilton, and B. Stumpf, "The File System of an Integrated Local Network", *Proceedings of the 1985 ACM Computer Science Conference*, March 1985, pp 309-324.

[Lamp 78] Lampson, B. and H. Sturgis, "Crash Recovery in Distributed Data Storage", To appear in CACM, (Xerox PARC report, 1978).

[Leu 86] Leu, P. and B. Bhargava, "Multidimenension Timestamp Protocol for Concurrency Control", *Proceedings of the IEEE 2nd Int. Conf. Data Engineering*, Los Angeles, CA, Feb 1986, pp 482-489.

[Lisk 83] Liskov, B. and R. Scheifler, "Guardians and Actions: Linguistic Support for Distributed Programs", *ACM TOPLAS*, vol 5, no. 3, pp 381-404, July 1983.

[Mino 82] Minoura, T. and G. Wiederhold, "Resilient Extended True-Copy Token Scheme for a Distributed Database System", *IEEE Trans. Software Eng.*, vol. SE-8, No. 3, 173-188, May 1982.

[Post 81] Postel, J., "Transmission Control Protocol - DARPA Internet Program Protocol Specification", *USC/Information Sciences Institute*, Sep 1981, RFC 793.

[Shri 82] Shrivastava, S. and F. Penzieri, "The Design of Reliable Remote Procedure call Mechanism", *IEEE Trans on Comp.*, Vol. 31(7), pp 692-697, July 1982.

[Tane 72] Tanenbaum, A., "Structure of Computer Organization", Prentice Hall, 1972

[Tane 81] Tanenbaum, A., "Computer Networks", Prentice Hall, 1981.

Description of Symbols

SS	stable storage: special I/O for log, backup info
LMM	low level memory management: simple physical memory services
UMM	upper level memory management: virtual memory, if desired; buffer pools
CO	communication: semi-reliable datagrams (at most once?) ; broadcast
LIO	low-level I/O: interface to hardware I/O
UIO	upper-level I/O: read/write protocol, permitting replication and providing location independence; includes logging
TR	transactions management: BeginTrans, EndTrans, and Abort verbs; provides concurrency control; must have 'hooks' in read/write system calls
RPC	remote procedure call support: reliable RPC
PC	partition control: provides virtual fail-proof network to upper layers (also handles site failure/recovery)

Figure 1

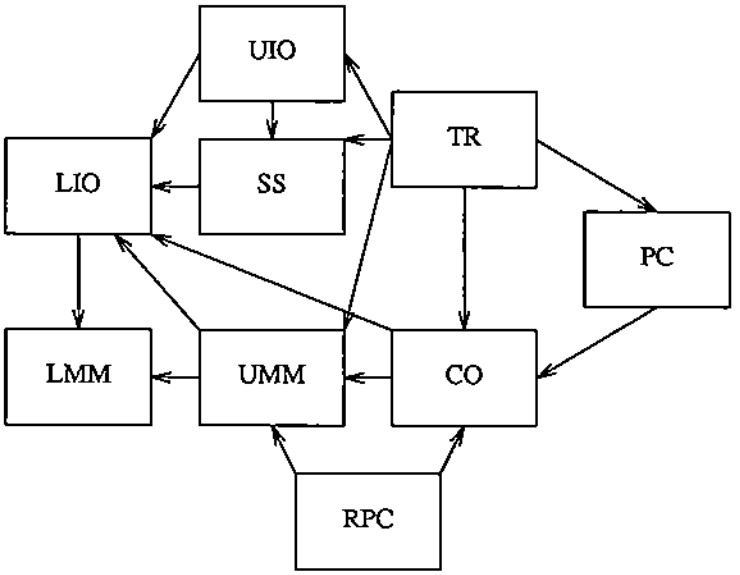

Figure 2: Dependency graph for proposed layered, distributed system.

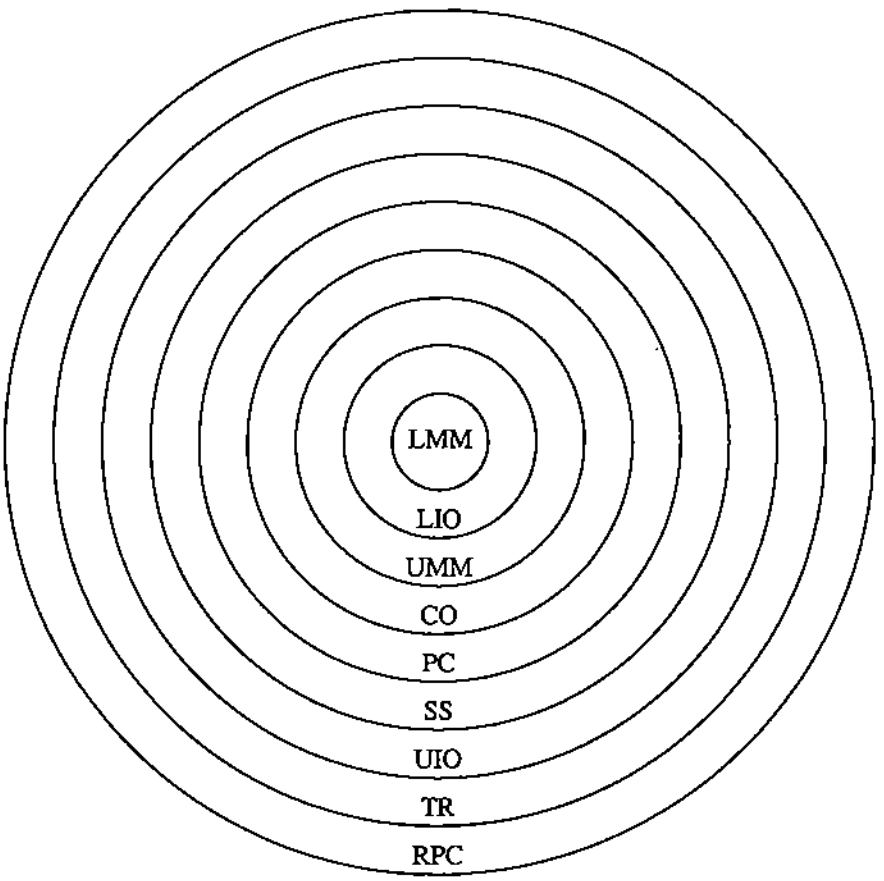

Figure 3: Possible layering, based on dependency graph.

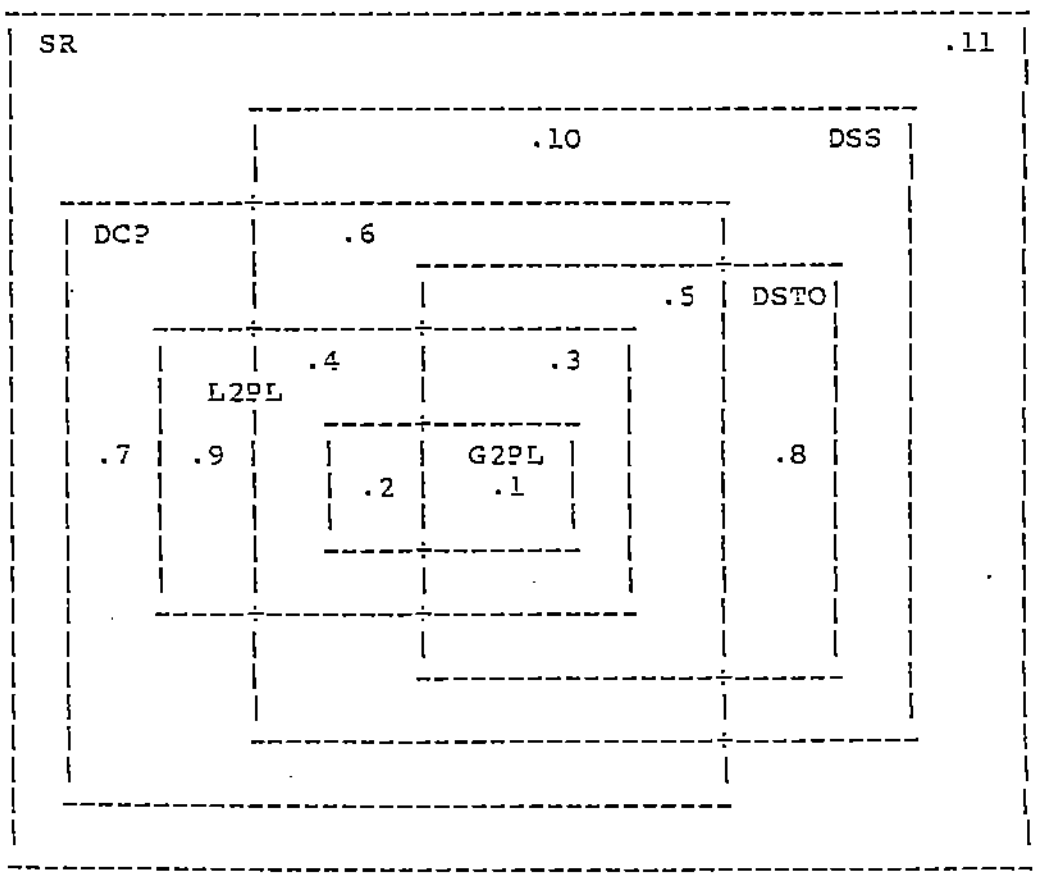

Figure 4: The hierarchy of the classes of concurrency control algorithms.

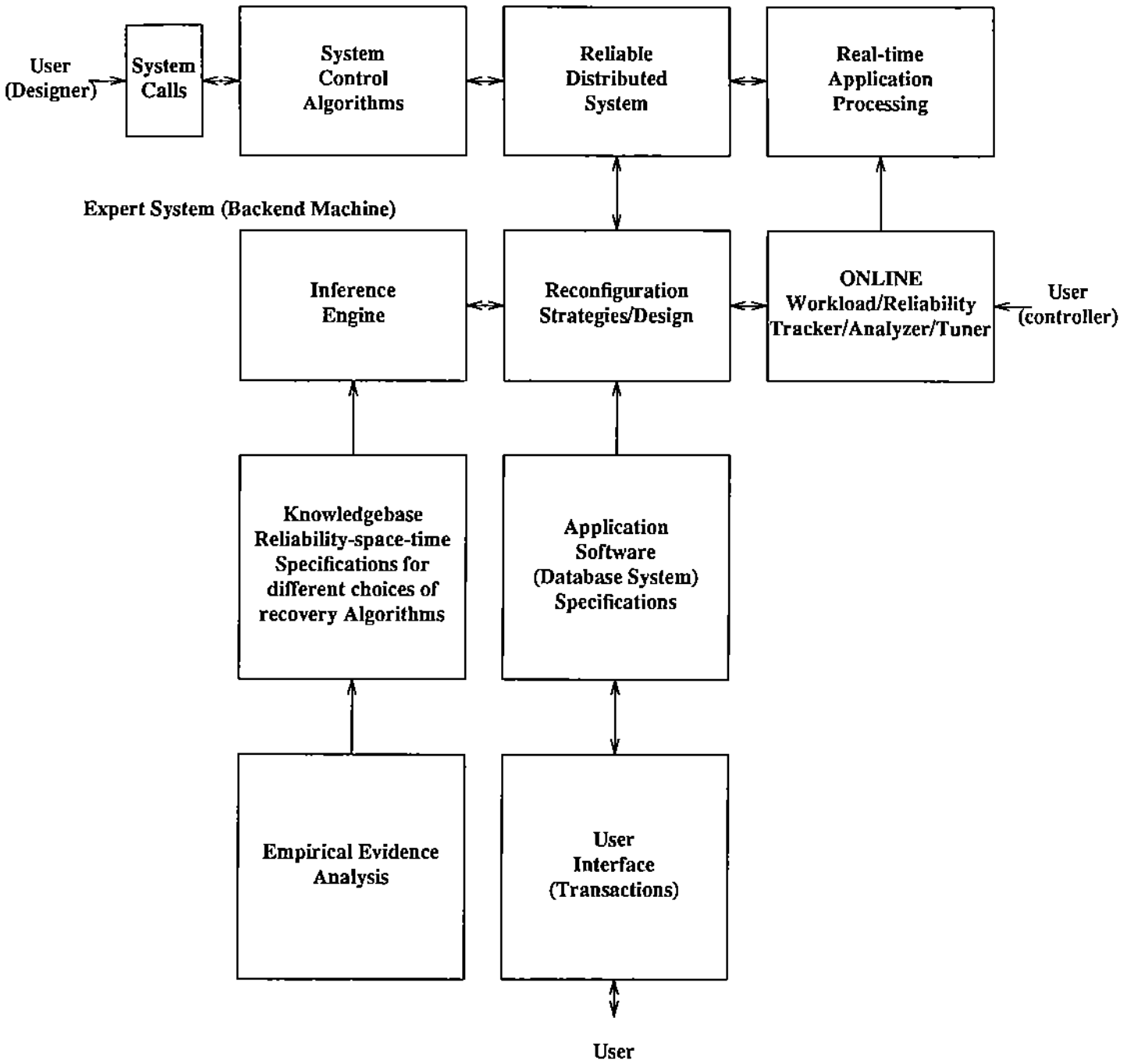

Figure 5: RAID-EXPERT

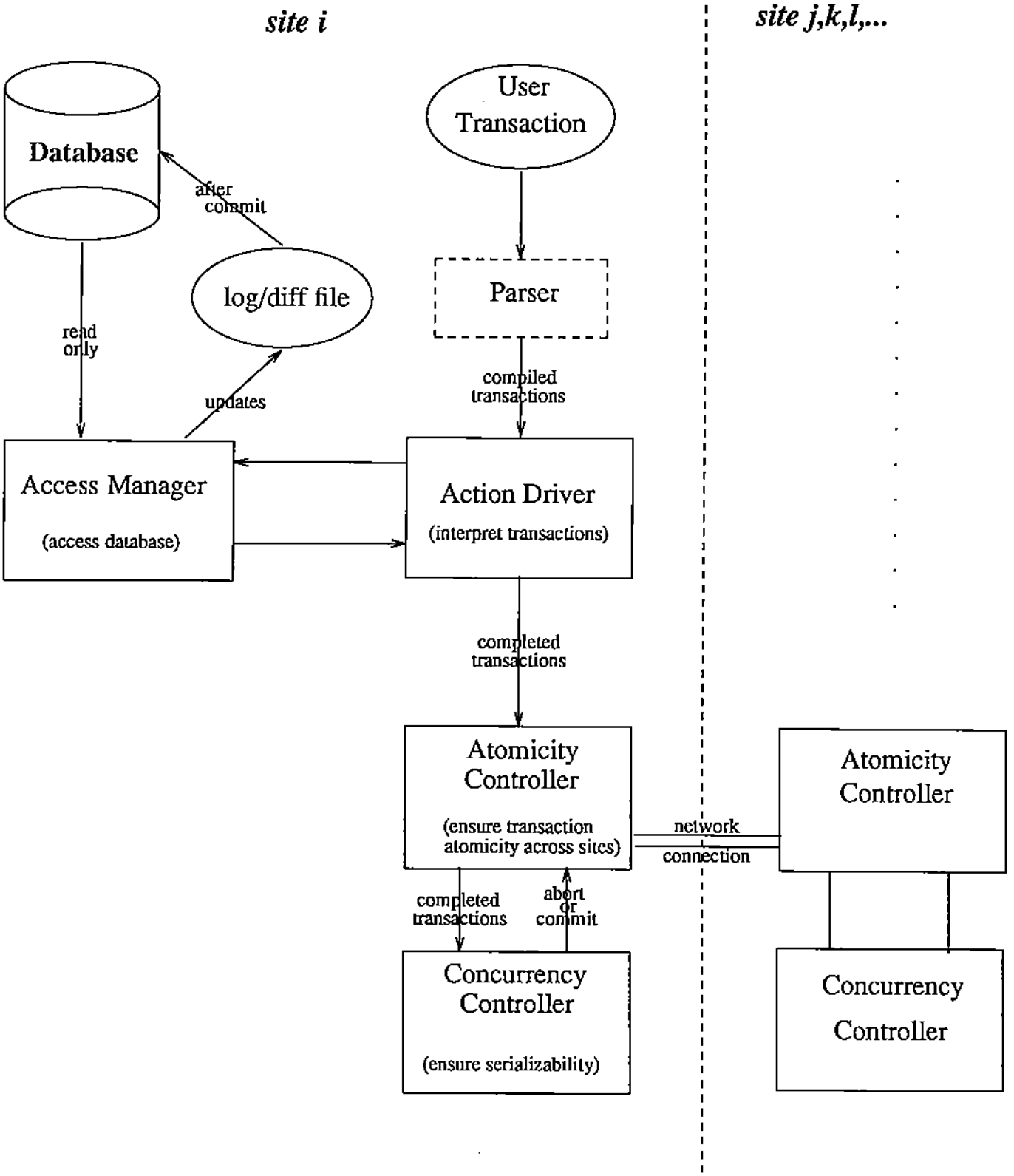

Figure 6: Structure of the RAID distributed system.

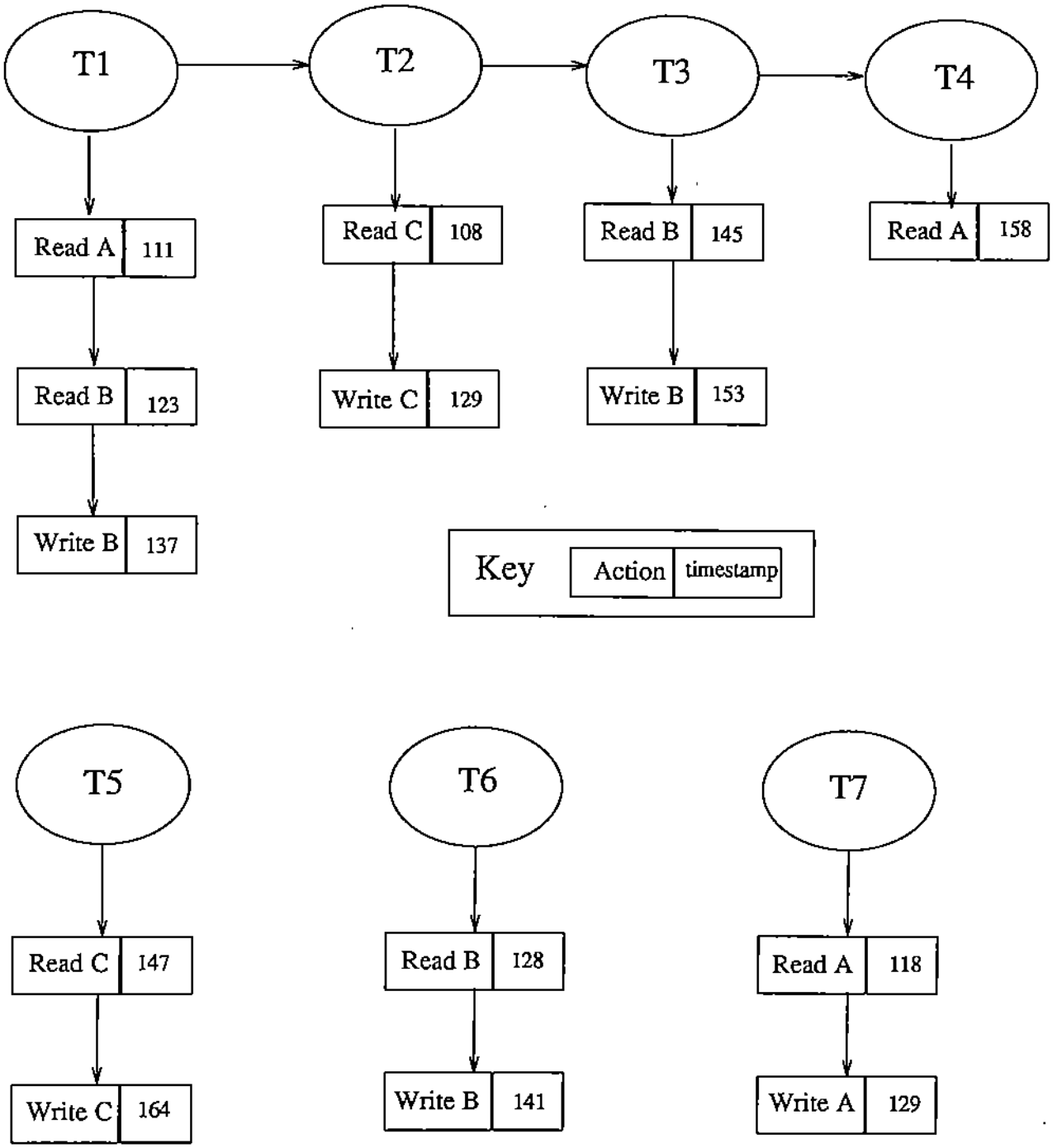

Figure 7: Example of general data structure for implementing timestamp/locking concurrency control protocols. T1 through T4 are a sample history of committed transactions. T5 is able to commit, and would be added to the history. T6 cannot be serialized with T1 through T5 and would be aborted. T7 would be aborted by a standard locking protocol, but is serializable with T1 through T5 and would be committed by some timestamp based protocols.

Explanation: New transaction N is attempting to commit. O is an old transaction against which N must be validated. O.A[D] refers to an access A (read/write) to data item D by transaction O. The simple locking protocol that we use here works by assigning a lock to a transaction when it first reads or writes a data item. All of its locks are released when it completes.

```
FOR each O: a transaction from the current history DO
  FOR each O.A[D] in O's action list DO
    FOR each N.A[D] in N's action list DO
      IF timestamp(O.A[D]) ≤ timestamp(N.A[D]) AND
        timestamp(O's completion) ≥ timestamp(N.A[D]) THEN
              ABORT transaction N
      FI
    OD
  OD
OD
```

Complexity

p = number of actions in this transaction
N = number of transactions in history
A = set of active transactions with time overlap with the new transaction
m_i = number of actions in T_i for $1 \leq i \leq N$

$$O(p \sum_{T_i \in A} m_i + N)$$

and uses space

$$O(\sum_{i=1}^{N} m_i + p).$$

If the action lists are likely to be large, performance can be improved by using a more sophisticated dictionary data structure such as a binary tree. Since action lists are likely to have high locality of reference, heuristics such as the move-to-front or transpose rules may be worthwhile.

Figure 8: An algorithm for enforcing simple locking as a validation protocol. In this algorithm a new transaction must not have violated any locks that would have been held by transactions in the existing history.

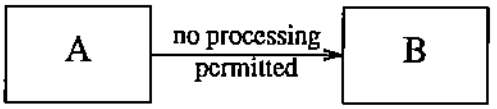

Figure 9: Naive approach to switching from concurrency control algorithm A to B.

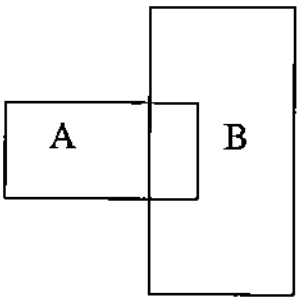

Figure 10: In this approach histories in the intersection of A and B are permitted.