There is a need to design distributed systems that are not rigid in their choice of algorithms and that are responsive to faults/failures and performance degradation. To meet this challenge, we formalize and experiment with design principles that allow the implementation of an adaptable distributed system. The strategies for dynamic reconfiguration of the subsystems and determining their impact are being studied via experiments on a prototype system called RAID under development at Purdue University. RAID provides system level support for transaction management in a reliable manner. Other transaction based systems are TABS [SBD*85], ARGUS [LS83], and System R* [LHM*84].

The key contribution of RAID is the system level support provided for building transaction based applications. RAID provides support for atomic objects and atomic commitment across a set of sites. It also includes concurrency control mechanisms based on time-stamps that provide a variety of choices of methods spanning from two-phase locking to optimistic methods utilizing the semantics of transactions and the objects accessed by them. In addition RAID has site failure and network partition control algorithms integrated with the rest of concurrent transaction processing and a replicated copy control subsystem.

1 Introduction

Current distributed systems provide a rigid choice of algorithms for implementation. For example, ARGUS has made the choice of using two phase locking as the concurrency control mechanism. TABS has decided to employ data servers for controlling concurrent access to objects and some type specific locking mechanisms are used. Since different applications need to be served by a distributed system, such design choices may not be appropriate universally. 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.

Our implementation effort has two main directions [BR86a]. First, each of the major sub-systems has been 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 have designed our system in layers with clearly defined communication paths between the layers. This will allow new protocols to fit into our system without disturbing protocols on other levels. Currently, our system is based on the following layers:

- **Low level memory management**: simple physical memory services
- **Low-level I/O**: interface to hardware I/O
- **Upper level memory management**: virtual memory, if desired; buffer pools
- **Communication**: reliable datagrams; broadcast
- **Partition control**: provides virtual fail-proof network to upper layers
- **Stable storage**: special I/O for logging and backup
- **Upper-level I/O**: read/write protocol, permitting replication and providing location independence; includes logging
- **Transaction management**: BeginTrans, EndTrans, and Abort verbs; provides concurrency control; must have 'hooks' in read/write system.
- **Remote procedure call support**: reliable RPC

2 Experimental System

RAID is an experimental distributed system being developed on VAXen and SUNs under the UNIX operating system. Figure 1 depicts the six major subsystems in RAID: Parser, Access Manager, Action Driver, Auditor (log/diff), Atomicity Controller, and Concurrency Controller. The auditor provides the implementation of atomic objects and works with the access manager to provide reliable I/O. The Parser accepts users' requests expressed in a relational calculus (INGRES-QUEL type) language and produces a transaction with several logical read/write actions. These actions are processed by the Action Driver which converts them into physical actions on the replicated copies of objects and communicates with the local Access Manager for I/O and with the local Atomicity Controller for commitment of transactions across the distributed system. The Atomicity Controller validates transactions for local serializability with the Concurrency Controller and communicates with other Atomicity Controllers using reliable broadcast to make the commitment decision.
A new module called Site Recovery has been recently implemented based on the idea that each site maintain a global vector representing the incarnation numbers of each site to deal with transactions in progress at the instance of site failure or during site recovery [BR86c]. We are optimistic about the feasibility of this approach since we have come across a report "Distributed System V IPC in LOCUS" by Brett Fleisch in which bootcount is used to identify current incarnation of the kernel. This module works closely with the Access Managers and the Atomicity Controllers.

Before posting the updates in the database, the Action Driver goes through the auditor that can use either a log or a differential-file based system. This mechanism provides the atomic object property. All sites in the system contain all six subsystems and can process local transactions independently and global transactions via the communications system that ties all the Atomicity Controllers together. The communications system is built on top of UDP/IP datagrams. Our system provides reliable arbitrary length asynchronous datagrams. Reliability is provided by piggybacking acknowledgments on the mandatory reply to a message. If a site does not respond within a timeout interval the site failure/recovery routine is invoked. An oracle provides location independent naming, although addresses are cached for efficiency. If the oracle crashes a new one is elected and all servers register with it.

Currently the system provides two choices for the auditor/back-up system and six choices for concurrency controller. Switching from one choice to another is currently done statically, although we are working on implementing dynamic switching. Part of our development effort is to find the kernel level services that are most useful in developing a distributed system and implementing them in UNIX.

3 Operating System Support

3.1 Support for Database Operations

Traditional time-sharing operating systems do not provide sufficient support for database processing [Sto81]. The problem is that efficient, reliable transaction processing has very different requirements from time-sharing users. This section discusses some of these differences, and suggests ways in which operating systems can support transaction processing without substantially increasing in size or complexity. We discuss these changes in terms of a database operating system (DBOS) which is an extension to an ordinary operating system designed especially to support database processing.

The kernel must offer an I/O interface that permits the development of stable storage and committed updates. A standard problem along these lines is that the kernel offers a nice caching scheme for efficiency which does not permit any guarantee that a write to the disk has actually taken place. At least the kernel must provide a flush function to force a write to non-volatile storage. Perhaps the DBOS should take over the job of managing I/O caching. Unfortunately, caching is needed for time-sharing systems also.

Because of the special nature of transaction processing, the DBOS needs a way to offer transactions especially efficient access to some resources. In particular, each transaction
acting on a large database should not incur a large operating system overhead for opening the database as a file. Either the DBOS should be able to give a hint to the operating system to cheat on file opens to particular files, or the DBOS will have to do the cheating itself. The DBOS needs a way to offer hints on access to memory also. This applies in two ways. First, traditional virtual memory page replacement schemes (e.g. LRU or FIFO) do not work well for transactions. The DBOS should be able to suggest pages to be replaced according to an algorithm of its own. Second, transactions will request lots of memory, probably in similarly sized chunks (e.g. the size of a database relation). It should be able to give the OS a hint about what sorts of memory requests to expect, both to improve speed and to avoid fragmentation. This sort of thing is already done in many operating systems for sub-systems such as the network interface.

Furthermore, the processor should be shared among DBOS processes in a special manner. It would probably be best for the kernel to allocate a (large) timeslice to the DBOS and then for the DBOS to suballocate it to processes. Perhaps this can be done by offering finer control than just priorities over CPU allocation (e.g. timeslice length; a special command to give up the CPU).

Note that the interaction between the additional control of the CPU, I/O, and memory sub-systems is likely to provide even more increased efficiency than any one of them individually. For instance, the DBOS can offer to give up a lot of pages just before swapping out, or force writes to disk while it is swapped out, etcetera.

### 3.2 Shared Memory

One of the temptations to distributed systems designers is to build major parts of their system within the kernel of an existing operating system. Two advantages of this approach are that it gives them access to especially efficient kernel level services (such as light-weight processes or inter-kernel communications protocols) and that it gives them an easy way to share data structures between different modules of their system. On the other hand, very large operating systems are often clumsy or unwieldy, become a significant drain on system resources and are unsuitable for tasks other than the one for which they were originally designed. Because of these disadvantages we are investigating ways in which operating systems can offer support that will allow efficient sub-systems – such as reliable distributed transaction systems – to be implemented outside of the kernel.

Figure 2 suggests four major approaches to this task. The first is based on operating system support for inter-process communication. This method suffers from inefficiency and low bandwidth communication. The second combines all of the modules of the distributed system in a single operating system process, which is responsible for sharing processor cycles between its various duties. This one has decreased modularity which is likely to decrease both reliability and adaptability, and is unable to schedule operations to take advantage of multi-processor hardware. The final two approaches depend on additional operating system support for shared memory. Support for shared segments allows only certain data structures to be shared (e.g. System V Unix shared segments), while a process group consists of a group of processes that have the same address space (e.g. teams
in V). Process groups are easier to implement, especially with some hardware support. Shared segments are the most general approach, offering full modularity, high bandwidth communication, and high efficiency.

4 Adaptability

We are developing a formal model of adaptation that offers guidelines for the successful development of adaptable protocols across a wide range of distributed algorithms. The most fruitful approach so far has been to separate the process of converting protocol A to protocol B into three regions. In the first region A is running as usual. In the second region processing is restricted to that permitted by both A and B, possibly with some additional restriction imposed by the conversion. Finally, in the third region the system has been converted to protocol B, which is running as usual. This approach provides a nice characterization of the conversion process that works for many distributed algorithms.

We have studied this model in the context of concurrency control, and have developed a general method for converting between concurrency control algorithms without halting processing [BR86b]. This method is easy and efficient to implement, and will allow adaptation of the concurrency controller in response to environmental measurements, such as multi-programming level, failure probability, etcetera.

5 Replicated Copy Control

One of the most important characteristics of a transaction based system is failure atomicity, the feature that either all or none of the updates of a transaction take place. A convenient implementation of this feature is to write the updates to a temporary workspace while the transaction is running, and then to post them to the database all at once when the transaction has committed. In a single site system these updates are usually written in a critical section so that no other transaction can observe or change the state between the commit decision and the time the updates actually appear in the database. The replicated copy control subsystem has the task of ensuring that transactions at all sites in the distributed system see the updates of committed transactions consistently in the presence of independent failures. Unfortunately, the single site solution of using a critical section is too expensive to be suitable for a distributed system.

A solution to this problem must guarantee that the concurrency controller is able to establish the correct read-from relationship for each distributed transaction. As long as this property is not violated, the serializability condition guaranteed by a correct concurrency controller will ensure that the state observed by each transaction is consistent. Unfortunately, this information is difficult to determine in a distributed concurrency controller, because the concurrency controller cannot rely on the time at which it decided to commit a transaction as an indication of the time at which the write was performed since communication may have delayed this write arbitrarily. However, one approach to solving the problem is suggested by this characterization. Each time an access to the object is
performed, the ID of the transaction doing the I/O is recorded with the object. Then the concurrency controller can correctly determine the reads-from relation. We have not implemented this idea yet but present it for discussion.

6 Conclusion

Our experimental work has already given us insight into the problems of the replicated copy control system. We found that the communication path between concurrency controllers and access managers via the atomicity controller is not going to work out when updates on different copies do not occur at approximately the same instance or interval. We are now restructuring the system and may possibly combine the access managers and the concurrency controllers to produce a module which may be similar to dataservers of TABS or guardians of ARGUS but will not necessarily use locking. Support for shared memory would allow for more a elegant structuring, so we are considering using a modified Unix kernel that supports shared memory. Initially our focus is on the database application but we are planning to use objects of various types and hope to try other applications after the system level support is fully worked out.

References


Figure 1: Structure of the RAID distributed system.

Figure 2: Process organization in Database Operating Systems.