STARLITE: AN ENVIRONMENT FOR DISTRIBUTED DATABASE PROTOTYPING

Sang H. Son
Jeremiah M. Ratner

Computer Science Report No. TR-89-05
August 2, 1989

StarLite: An Environment for Distributed Database Prototyping

Sang H. Son
Jeremiah M. Ratner

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

One of the reasons for the difficulty in evaluating new techniques for distributed database systems is the complexity involved due to a large number of system parameters that may change dynamically. Prototyping methods can be applied effectively to the evaluation of database techniques. In addition, database technology can be implemented in a modular reusable form to enhance experimentation. This paper presents a software prototyping environment for the development and evaluation of distributed database systems. The prototyping environment is based on concurrent programming kernel which supports the creation, blocking, and termination of processes, as well as scheduling and inter-process communication. The paper describes the the structure of a prototyping environment and a series of experimentation performed, using the environment, for performance evaluation of concurrency control algorithms. One of the key aspects of the prototyping environment is that software, which was developed with the prototyping environment, can be easily ported to a target hardware system for embedded testing.

Index Terms - distributed systems, prototyping, synchronization, transaction, communication, port

---

This work was supported in part by the Office of Naval Research under contract number N00014-88-K-0245, by the Center for Innovative Technology under contract number CAE-89-007, and by the Federal Systems Division of IBM Corporation under University Agreement WG-249153.

1. Introduction

The advantages of distributed database systems over a single-node database system are apparent. Queries may be distributed among the nodes of the system and processed concurrently. They also provide higher reliability and availability through replication of critical data and resources. In addition, new nodes and capacities can be added in a modular fashion without disturbing ongoing work and with only marginal investment. The availability of new high-performance communication technology makes a distributed database system on a specialized LAN more attractive than ever.

As the advantages of distributed database systems become more apparent, so does the necessity of developing reliable, high-performance distributed systems as cheaply and quickly as possible. However, distributed system development has been hampered by a lack of available tools in both the design and implementation phases. The additional complexity of developing distributed system software and insuring specified performance has lengthened the time required to bring a distributed system on-line and hampered efforts to quantitatively verify performance and reliability characteristics of the algorithms for distributed systems.

A prototyping technique can be applied effectively to the evaluation of new techniques for distributed database systems. A *database prototyping environment* is a software package that supports the investigation of the properties of a database control techniques in an environment other than that of the target database system. The advantages of an environment that provides prototyping capability are obvious. First, it is cost effective. If experiments for a twenty-node distributed database system can be executed in a software environment, it is not necessary to purchase a twenty-node distributed system, reducing the cost of evaluating design alternatives. Second, design alternatives can be evaluated in a uniform environment with the same system parameters, making a fair comparison. Finally, as technology changes, the environment need only be updated to provide researchers with the ability to perform new experiments.

A prototyping environment can reduce the time of evaluating new technologies and design alternatives. From our past experience, we assume that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms, while the majority of code deals with intrinsic problems, such as file management. Thus, by properly isolating technology-dependent portions of a database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation.

A distributed prototyping environment implies additional design issues not present in centralized prototyping systems. The communication subsystem, for example, must be completely functional and yet general enough to allow for the execution of various protocols with their specific overheads. In addition, the environment must be error free, yet allow for artificial insertion of node failures and errors in communication to enable testing of distributed software reliability.

Another design requirement of prototyping is that performance measurement be part of the environment, so that the relative efficiency of various systems may be measured through timing simulation without actual implementation on hardware. However, unlike a centralized environment, the distributed prototyping environment does not support a single global time, and timing information is available only locally.

In addition, the environment must be designed to be flexible in allowing various systems and system parameters to be represented. For this criterion, the difficulty lies in developing operational but extensible interfaces that allow the functions and data objects presented to the user to be mapped onto particular modules with special characteristics. It requires the structure of data types and the implementation of functions to be hidden within the modules that export them. In this way, individual modules can be augmented "horizontally", without affecting the rest of the implementation, by embedding within their functions and types those operations or extensions that represent parameters and/or overheads of a particular system.

This paper describes a message-based approach to prototyping study of distributed database systems, and presents a prototyping software implemented for a series of experiments to evaluate concurrency control algorithms in a distributed environment.

2. Message-Based Prototyping

For a message-based prototyping, the process view of a system has been adopted. A distributed system being prototyped consists of a number of *processes* which interact with others at discrete instants of time. Processes are basic building blocks of a prototyping environment. A process is an independent, dynamic entity which manipulate *resources* to achieve its objectives. A resource is a passive object and may be represented by a simple variable or a complex data structure. A prototyping environment models the dynamic behavior of processes, resources, and their interactions as they evolve in time. Each physical operation of the system is modeled by a process, and the process interactions are called *events*.

We use the client/server paradigm for process interaction in our prototype. The system consists of a set of clients and servers, which are processes that cooperate for the purpose of transaction processing. Each server provides a service to the clients of the system, where a client can request a service by sending a request message (a message of type *request)* to the corresponding server. The computation structure of the system to be modeled can be characterized by the way clients and servers are mapped into processes. For example, a server might consists of a fixed number of processes, each of which may execute requests from every transaction, or it might consists of varying number of processes, each of which executes on behalf of exactly one transaction.

A message-based prototyping environment, if the modularity is "right", can be of enormous benefit in designing and testing emerging systems and in comparing and improving algorithms that are applicable to many different systems. Some of the benefits that may be achieved by using a prototyping environment are discussed in the following.

2.1. System Development

One of the benefits of a prototyping environment is that the software to be used in an actual system can be developed using the environment. The prototyping environment can support a simulated environment, actual hardware, or a "hybrid" mode in which some of the modules are implemented in hardware and some simulated. In this way, it is irrelevant to the software developed in the environment whether or not all or part of the environment is running on hardware.

It is very common that much of the codes in new systems is the same as in previous systems. The modularity of the prototyping environment enables the isolation of only those aspects of existing systems that need to be changed. In addition, design alternatives for those aspects can be implemented and evaluated rapidly.

Another important use of a prototyping environment is to analyze the reliability of database control mechanisms and techniques. Since distributed systems are expected to work correctly under various failure situations, the behavior of distributed database systems in degraded circumstances needs to be well understood. Although new approaches for synchronization, checkpointing, and database recovery have been developed recently [Son87, Son88, Son89], experimentation to verify their reliability properties and to evaluate their performance has not been performed due to the lack of appropriate test tools.

The environment can be programmed to lose a specified percentage of messages, and specified sites can be crashed and recovered at the user's request. Thus the reliability of a piece of software can be tested easily and quickly under dramatically different levels of hardware reliability. In addition, some random elements in tests can be controlled. By using the same seed when generating random events, test situations can be repeated precisely. If system fails under certain conditions, it is possible to discover the flaw that enabled the failure by repeating the test with the same parameters. This is very hard to achieve when testing in a hardware environment.

2.2. Algorithm Development

When an algorithm is being developed, most often we can identify a number of parameters that will affect its performance. However, it is very hard to know which factor is dominating, and its sensitivity. A prototyping environment can be used to investigate the behavior of a given algorithm, and compare it with other algorithms, under various system parameters: size, configuration, speed, reliability, etc.

Consider data replication, for instance. As mentioned above, replication is one of the advantages of distributed systems. However, replication complicates the tasks of synchronizing data and ensuring mutual consistency. One replication control algorithm has advantages over the others under certain reliability conditions and system parameters, but disadvantages under other conditions. Indeed, some authors acknowledge the limitations of their algorithms [Bern84]. Of interest would be to quantify the performance, reliability, and predictability penalties suffered by each algorithm when conditions favorable to them are changed. In addition, reasons for degraded performance can be analyzed and the algorithms can be extended to handle exceptional conditions.

Another example is in the area of concurrency control algorithms. For instance, an interesting optimistic concurrency control algorithm that reduces the probability of abort is proposed in [Boks87]. It works by maintaining, for each transaction, an acceptable range from which the transaction may choose its final, discrete timestamp. However, this requires that every time a transaction commits and selects a discrete timestamp, the timestamp ranges of all other active transactions must be adjusted in order to maintain some serializable execution order. The tradeoff between fewer aborts and the overhead imposed by this adjustment is difficult to assess by analytic modeling. The prototyping environment can be used effectively for evaluating this algorithm and comparing it with other concurrency control algorithms.

3. Structure of the Prototyping Environment

For a prototyping tool for distributed database systems to be effective, appropriate operating system support is mandatory. Database control mechanisms need to be integrated with the operating system, because the correct functioning of control algorithms depends on the services of the underlying operating

system; therefore, an integrated design reduces the significant overhead of a layered approach during execution.

Although an integrated approach is desirable, the system needs to support flexibility which may not be possible in an integrated approach. In this regard, the concept of developing a library of modules with different performance and reliability characteristics for an operating system as well as database control functions seems promising. Our prototyping environment follows this approach [Cook87, Son88b]. It is designed as a modular, message-passing system to support easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. It efficiently supports a spectrum of distributed database functions at the operating system level, and facilitates the construction of multiple "views" with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for new system setup.

The prototyping environment provides support for transaction processing, including transparency to concurrent access, data distribution, and atomicity. An instance of the prototyping environment can manage any number of virtual sites specified by the user. Modules that implement transaction processing are decomposed into several server processes, and they communicate among themselves through ports. The clean interface between server processes simplifies incorporating new algorithms and facilities into the prototyping environment, or testing alternate implementations of algorithms. To permit concurrent transactions on a single site, there is a separate process for each transaction that coordinates with other server processes.

Figure 1 illustrates the structure of the prototyping environment. The prototyping environment is based on a concurrent programming kernel, called the StarLite kernel. The StarLite kernel supports process control to create, ready, block, and terminate processes. Scheduler in the kernel maintains a virtual clock and provides the **hold** primitive to control the passage of time. The benefit of a virtual clock is that any number of performance monitoring operations may be performed at an instant of virtual time. If a

physical clock were embedded, the monitoring activities themselves would interfere with other system activities and add to the execution time, resulting in incorrect performance measures.

User Interface (UI) is a front-end invoked when the prototyping environment begins. UI is menu-driven, and designed to be flexible in allowing users to experiment various configurations with different system parameters. A user can specify the following:

- system configuration: number of sites and the number of server processes at each site, topology and communication costs.
- database configuration: database at each site with user defined structure, size, granularity, and levels of replication.

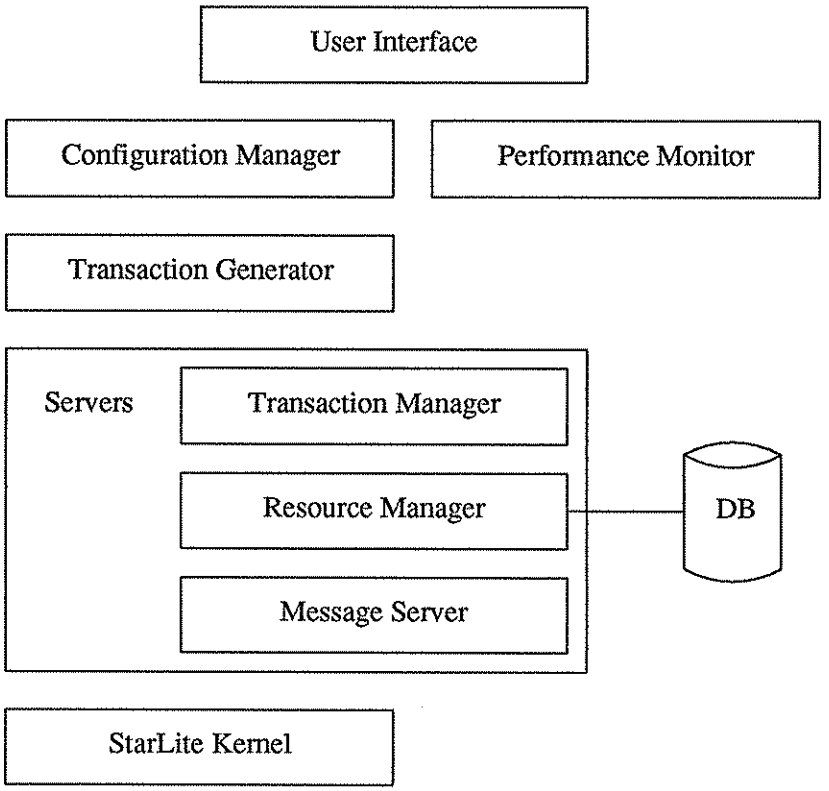

Fig. 1. Structure of the prototyping environment

- load characteristics: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only or update) and their priorities, and the mean interarrival time of transactions.
- concurrency control: locking, timestamp ordering, and priority-based.

UI initiates the Configuration Manager (CM) which initializes necessary data structures for transaction processing based on user specification. CM invokes the Transaction Generator at an appropriate time interval to generate the next transaction to form a Poisson process of transaction arrival. When a transaction is generated, it is assigned an identifier that is unique among all transactions in the system.

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. Each transaction is assigned to the Transaction Manager (TM). TM issues service requests on behalf of the transaction and reacts appropriately to the request replies. For instance, if a transaction requests access to a file and that file is locked, TM executes either blocking operation to wait until the data object can be accessed, or aborting procedure, depending on the situation. If granting access to a resource will produce deadlock, TM receives an abort response and aborts the transaction. Transactions commit in two phases. The first commit phase consists of at least one round of messages to determine if the transaction can be globally committed. Additional rounds may be used to handle potential failures. The second commit phase causes the data objects to be written to the database for successful transactions. TM executes the two commit phases to ensure that a transaction commits or aborts globally.

The Message Server (MS) is a process listening on a well-known port for messages from remote sites. When a message is sent to a remote site, it is placed on the message queue of the destination site and the sender blocks itself on a private semaphore until the message is retrieved by MS. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When MS retrieves a message, it wakes the sender process and forwards the message to the proper servers or TM. The prototyping environment implements Ada-style rendezvous (synchronous) as well as asynchronous message passing. Inter-process communication within a site does not go through the Message Server; processes send and

receive messages directly through their associated ports.

The inter-process communication structure is designed to provide a simple and flexible interface to the client processes of the application software, independent from the low-level hardware configurations. It is split into three levels of hierarchy, as shown in Figure 2.

The Transport layer is the interface to the application software, thus it is designed to be as abstract as possible in order to support different port structures and various message types. In addition, application level processes need not know the details of the destination device. The invariant built into the design of the inter-process communication interface is that the application level sender allocates the space for a

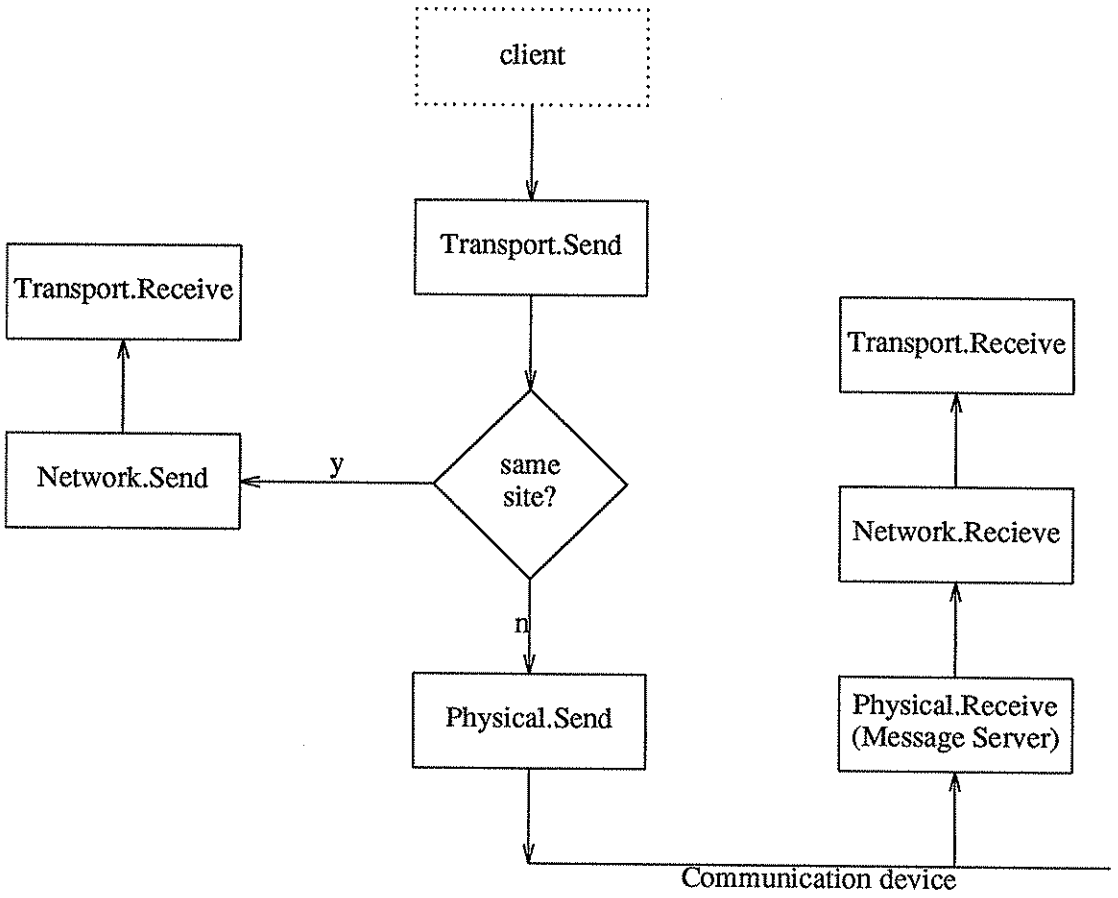

Fig. 2. Inter-Process Communication

message, and the receiver deallocates it. Thus, it is irrelevant whether or not the sender and receiver share memory space, i.e., whether or not the Physical layer on the sender's side copies the message into a buffer and deallocates it at the sender's site, and the Physical layer at the receiver's site allocates space for the message. This enables prototyping distributed systems or multiprocessors with no shared memory, as well as multiprocessors with shared memory space. When prototyping the latter, only addresses need to be passed in messages without intermediate allocation and deallocation.

The Physical layer of message passing simulates the physical sending and receiving of bits over a communication medium, i.e., it is for intersite message passing. The device number in the interface is simply a cardinal number; this enables the implementation to be simple and extensible enough to support any application. To simulate sending or to actually send over an Ethernet in the target system, for example, a module could map network addresses onto cardinals. To send from one processor to another in a multiprocessor or in a distributed system, the cardinals can represent processor numbers.

Messages are passed to specific processes at specific sites in the Network layer of the communications interface. This layer serves to separate the Transport and the Physical layers, so that the Transport layer interface can be processor- and process-independent and the Physical layer interface need be concerned only with the sending of bits from one site to another. The Transport layer interface of the communication subsystem is implemented in the Transport module. A Transport-level Send is made to an abstraction called a *PortTag*. This abstraction is advantageous because the implementation (i.e., what a PortTag represents) is hidden in the Ports module. Thus the PortTag can be mapped onto any port structure or the reception points of any other message passing system. The Transport-level Send operation builds a packet consisting of the sender's PortTag, used for replies, the destination PortTag, and the address of the message. It then retrieves from the destination PortTag the destination device number. If this number is the same as the sender's, the Send is an intra-site message communication, and hence the Network-level Send is performed. Otherwise the send requires the Physical module for intersite communication.

Note that accesses to the implementation details of the PortTag are restricted to the module that actually implements it; this enables changing the implementation without recompiling the rest of the system. As shown in Figure 3, the Transport module, which is the highest-level interface of the inter-process communication structure for the client processes, is very simple and elegant, and it achieves the desired flexibility.

The Performance Monitor interacts with the transaction managers to record, priority/timestamp and read/write data set for each transaction, time when each event occurred, statistics for each transaction and cpu hold interval in each node. The statistics for a transaction includes arrival time, start time, total processing time, blocked interval, whether deadline was missed or not, and the number of aborts.

Since each TM is a separate process, each has its own data area in which to keep track of the time when a service request is sent out and the time the response arrives, as well as the time when a transaction begins blocking, waiting for a resource, and the time the resource is granted. When a transaction com-

```
DEFINITION MODULE Transport;

FROM SYSTEM IMPORT BYTE, ADDRESS;
FROM Ports IMPORT PortTag;

PROCEDURE Send(pt : PortTag; VAR mess : ARRAY OF BYTE);
(* This Send is the highest-level, the transaction's interface.          *)
(* Send inverts PortTag to <processor number, Port number>.              *)
(* If the processor number is different from the sender's,               *)
(* Physical level Send is called, which sends to a different site.       *)

PROCEDURE Receive(pt : PortTag) : ADDRESS;
(* This routine is called by the destination process.                    *)
(* If the message has not arrived at the port associated with the PortTag,  *)
(* the destination process will be blocked.                              *)
(* When a message arrives at that port, this routine returns             *)
(* the address of the message.                                           *)

PROCEDURE NonblockReceive(pt : PortTag) : ADDRESS;
(* This routine is a non-blocking version of the Receive explained above.  *)
(* The destination process is not blocked if there is no message.        *)

END Transport.
```

Fig. 3. Transport Module

mits, it calls a procedure that records the above measures; when the simulation clock has expired, these measures are printed out for all transactions.

4. Experiments: Prototyping a Distributed Database System

The previous section described the structure of the prototyping environment. In this section, we present a distributed database system implemented using the prototyping environment. Two goals of our prototyping work were 1) evaluation of the prototyping environment itself in terms of correctness, functionality, and modularity, by using it in implementing distributed database systems, and 2) performance comparison between two-phase locking (2PL) and basic timestamp ordering (BTO) synchronization algorithms through the sensitivity study of key parameters that affect performance.

4.1. Model

An important component of performance evaluation is the assumptions used in system implementation. In our experiments, transaction are generated with exponentially distributed interarrival times, and the data objects updated by a transaction are chosen uniformly from the database. Two groups of transactions with different characteristics (e.g., type and number of access to data objects) are executed concurrently in order to aid in the modeling of realistic workload. A transaction has an execution profile which alternates data access requests with equal computation requests, and some processing requirement for termination (either commit or abort). Thus the total processing time of a transaction is directly related to the number of data objects accessed.

Both implementations are strict, that is, no transaction reads or writes a data object until all transactions that have written the object are committed or aborted [Ber87]. We assume that the time required to compare timestamps is equal to the time required to set a lock [Car86].

Accessing a data object requires an I/O cost and a CPU cost to process it. For modeling convenience, it is assumed that all reads precede all writes. This implies that all reads access the certified object value, not the value written by the transaction in its private workspace. The cost of accessing the most

recent committed version of an object is therefore always one disk access. In addition, it is assumed that the data is stored entirely on disk, and the data object size matches the disk block size, so that each disk access requires only one page access. Modeling a database stored partially or entirely in main memory would require adjusting these delays. We also assume that no transaction reads or writes the same data object more than once.

We assume that one simulation time unit equals 1 millisecond of actual time. It is the unit cost for any activity in the system; comparing timestamps or looking for a lock table takes 1 millisecond. In addition, a disk access is assumed to take 35 milliseconds of I/O time and 10 milliseconds of CPU time [Car86]. Internode communication delay is set through user-specified parameters. In all the experiments below, we set communication delay between any two directly connected nodes at 8 time units.

4.2. Experiment 1: Connectivity

Our first experiment investigates the effect of the level of connectivity of the system on average response time of transactions. The environment is an completely interconnected 8-node system with 500 unreplicated data objects uniformly distributed. The transaction mix is 75% read-only transactions and 25% updates. Each read-only transaction accesses 15 data objects and the update transactions access 5 data objects. The updates read each data object before writing to it, e.g., there are no "blind writes." The multiprogramming level is 10, i.e., there are always 10 active transactions executing, even though some of them may be blocked. We change the level of connectivity from a ring, i.e., each node having 2 neighbors, to completely interconnected, i.e, each node having 7 neighbors.

As shown in Figure 4, higher degrees of connectivity in the system result in lower average response times for both concurrency control mechanisms. This is to be expected, since both mechanisms require at least two messages per access, and possibly more in the case of access denial or releasing of locks. Therefore, lower communication costs will have a significant effect on response time.

We also see that BTO outperforms 2PL in all but the fully interconnected case. This is because when data conflict is low, the dominating factor in average response time is communication cost. The

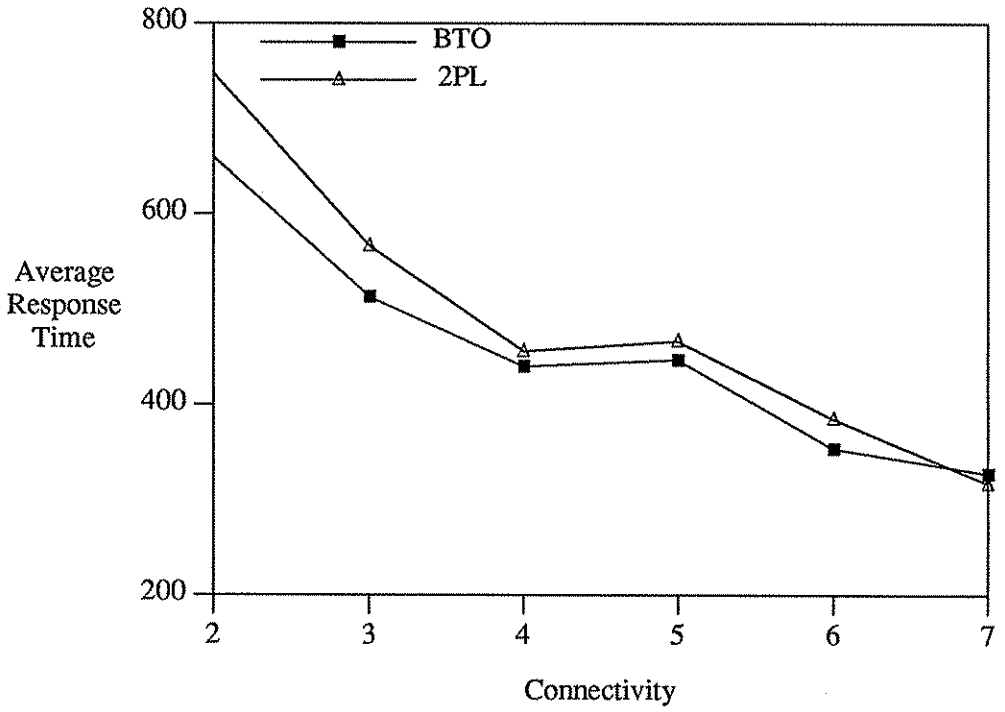

Figure 4. Response time sensitivity to connectivity

communication costs associated with 2PL are higher, because when a transaction commits, regardless of its access type, it must release all its read and write locks. For instance, when a read-only transaction commits in a BTO-based system, it incurs no overhead at all before termination, whereas the same transaction in a 2PL system still must release all its read-locks before termination. This requires extra communication costs as well as some extra processing costs. Note, however, that in a fully connected situation, 2PL outperforms BTO. This is because the communication costs are minimized by allowing each node to communicate directly with any other node, and the advantage of 2PL in dealing with long reads pays off. This advantage of 2PL will be discussed further in the next experiment. Since 75% of the transactions in this example are reads, and the reads are 3 times as long as the updates, the mechanism that handles longer reads more efficiently will produce the better results, given that communication cost is not very high.

4.3. Experiment 2: Size of Read-only Transaction

In this experiment, we change the number of objects accessed by each read-only transaction to range from 3% to 8% of the database. The update transaction size is again 1% of the database. All other parameters are as they were in Experiment 1. As shown in Figure 5, 2PL outperforms BTO as the size of read-only transactions gets bigger. This can be explained as the following.

First, BTO is biased against large read-only transactions, and 2PL is biased against small updates. In BTO, if a transaction attempts to read an object out of timestamp order, i.e., when the transaction's timestamp is smaller than that of the object, the transaction aborts and restarts. The probability of such an event increases as the read-only transactions access more data objects with a constant level of updates simultaneously in the system. Thus the probability of a read-only transaction aborting and restarting increases along with the size of its access set. Two-phase locking, on the other hand, allows long read-only transaction to lock their data sets for a long time, even if they sometimes have to wait for updates to release conflicting locks. Short update transactions must wait about half the response time of the read-only transaction, on the average, to access data objects. Since read requests dominate in this experiment, the algorithm that handles them more efficiently performs better.

Second, expanding the access set of read-only transactions while keeping the updates short will increase the probability of abort in the BTO algorithm. Figure 6 shows the execution sequence of a relatively long read (T_1) and a short update (T_2). If T_1 reads x after T_2 commits, T_1 will abort. If the update is short, the "window of vulnerability" of T_1 after T_2 commits is relatively wide, making it more probable that T_1 will read an object out of order and abort.

In 2PL, transactions are aborted only in case of deadlock. All deadlock cycles require at least one update transaction, and since most transactions in this experiment are reads, deadlock will most likely involve only a single update. The probability of deadlock involving a read-only transaction and an update transaction is low, since the "window of vulnerability" of a deadlock extends only from the time the update transaction requests a write lock until the end of the update transaction. This is a much shorter

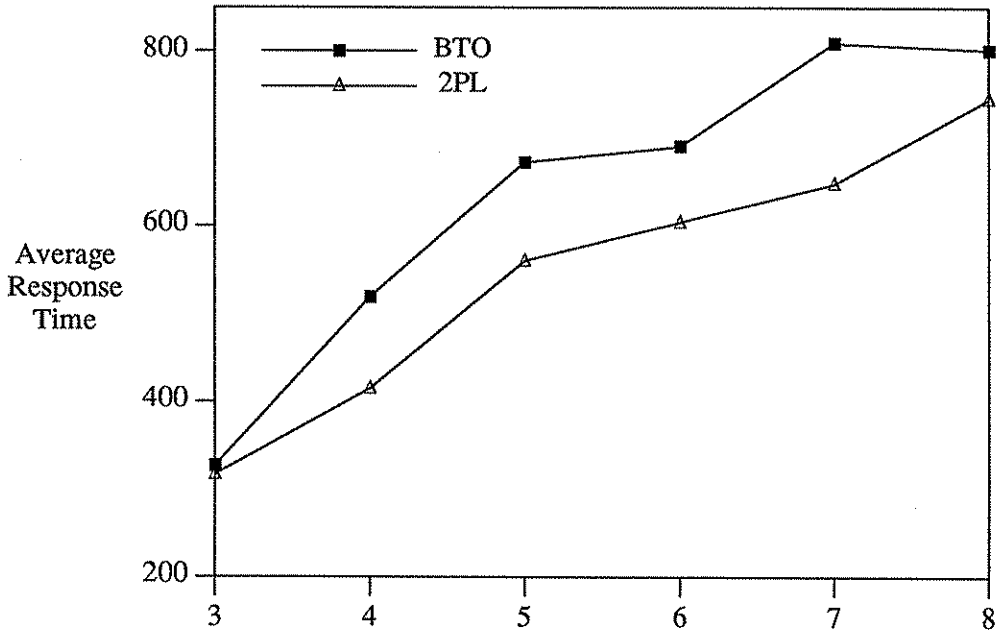

Figure 5. Response time sensitivity to read-only transaction size

period than in BTO, where the window extends from the end of the short update transaction to the end of the long read-only transaction.

When a transaction aborts, it delays 1 average response time before restarting. This adaptive restart delay has a stabilizing effect on response time, since it gives the conflicting transaction time to get out of the system [Agr85]. However, because of this delay in restart, increase in aborts drives up the average

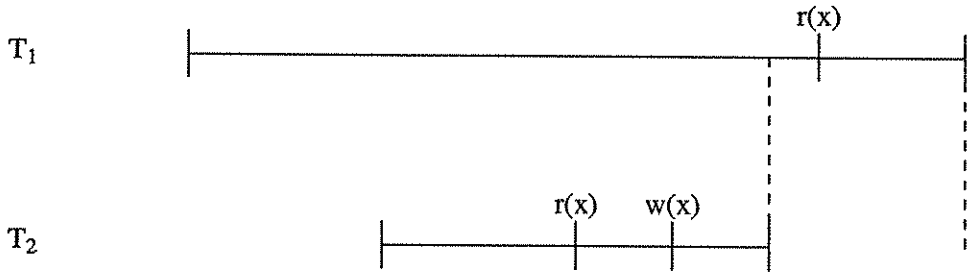

Figure 6. BTO bias against long reads

response time very rapidly. Aborts are even more expensive in a distributed environment than in a centralized environment, since requests to release resources hold by the aborting transaction must be sent all over the system.

Note that the average response time using BTO is stablized as the read-only transactions get quite long, in the range of 8% of the database. This is not because BTO has finally adapted well to the larger reads, but because this is an *average* response time, including the updates. In Figure 7, we see that throughput decreases as the read size grows from 7% to 8% of the database, even though the average response time using BTO declines over the same interval. When the read sizes are quite large, so many of the reads abort and are in the restart phase that the actual multiprogramming level declines. In other words, even if the multiprogramming level is 10, actual multiprogramming level is much lower since many of them are read-only transactions in the restart delay phase. Update transactions benefit from the reduced level of multiprogramming, thus reducing the overall average response time. Here we see a situation in which an increase in the level of *data contention* leads to a decrease in the level of *resource contention* [Bern87].

4.4. Experiment 3: Percentage of Read-Only Transactions

In this experiment, we investigate the impact of the percentage of read-only transactions on system performance. We run the experiment on a fully connected 8 node system. Each read-only transaction accesses 6% of the database and each update transaction accesses 2%.

The results are shown in Figure 8. When the percentage of reads is small, say 25%, deadlock probability is high in 2PL, resulting in long restart delay. As discussed in Experiment 2, this effect becomes the dominating factor in determining average response time. In BTO, however, two concurrent updates to the same data object not necessarily require one of them to abort. Even if the later update has a smaller timestamp than the previous update, it may not abort by applying the *Thomas Write Rule* [Bern87]. Thus the number of aborts is smaller in BTO, resulting in a shorter response time.

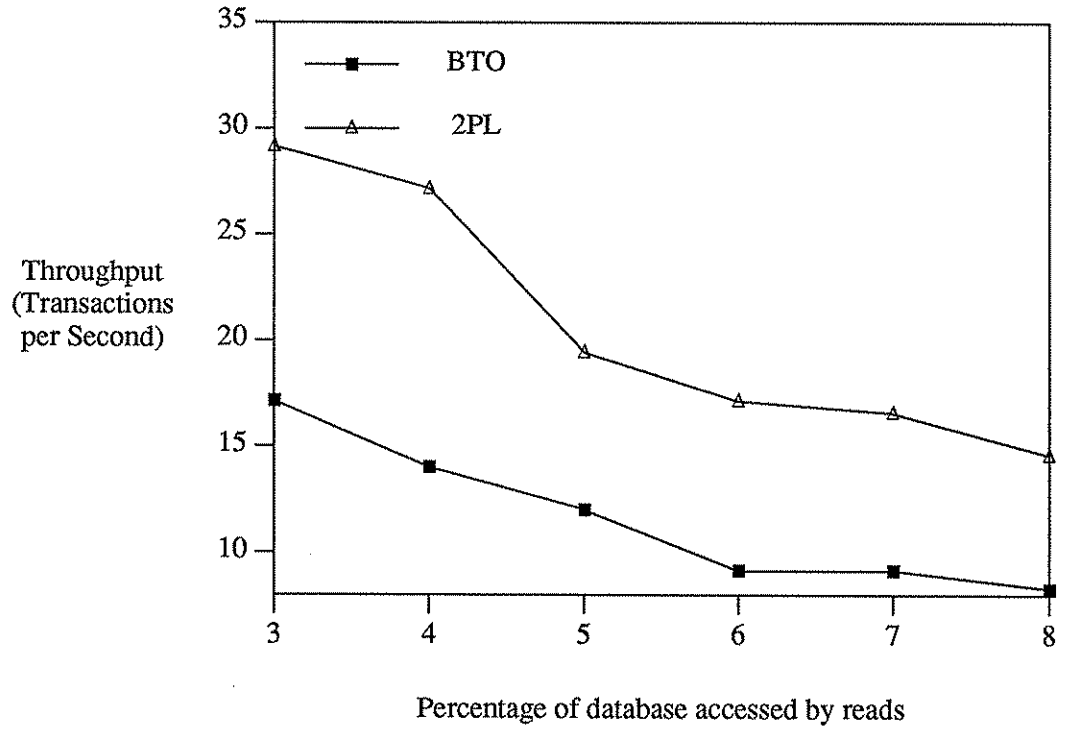

Figure 7. Throughput sensitivity to percent of database accessed by reads

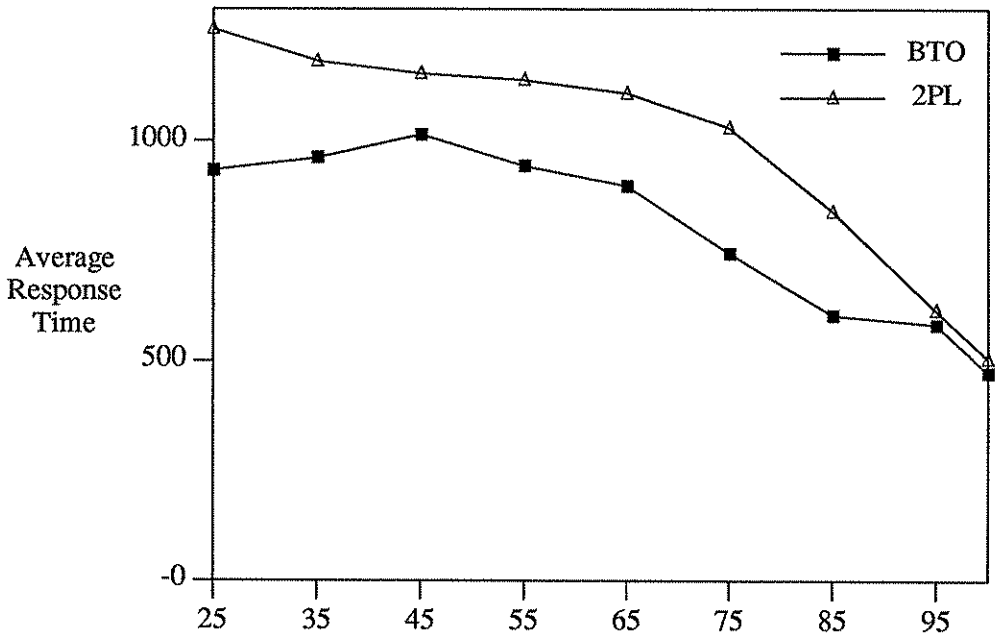

Figure 8. Response time sensitivity to percent of read-only transactions

Difference in throughput is more significant than that in response time. Difference in response time ranges from 5-40%, however, as shown in Figure 9, difference in throughput is about 100%, until the transaction mix is almost all reads. This shows that more computation time in 2PL is being spent on transactions that later abort.

Another factor in longer response time in 2PL is its bias against update transactions in the face of long reads. Even at 95-100% reads, BTO still slightly outperforms 2PL. As discussed in Experiment 1, this is due to the extra overhead for reads in 2PL when transactions commit, since they must send unlock requests over the system. When the data conflict level is low, this added communication cost becomes the distinguishing factor between the two concurrency control mechanisms.

One might expect that as the percentage of reads gets larger, the bias of BTO against long reads would take over as the dominant factor, and hence 2PL would outperform BTO. However, in this experiment the percentage of the database accessed by each update is fairly large, 2% as opposed to 1% in Experiment 2. Therefore, the probability of deadlock is still fairly high and tends to dominate the

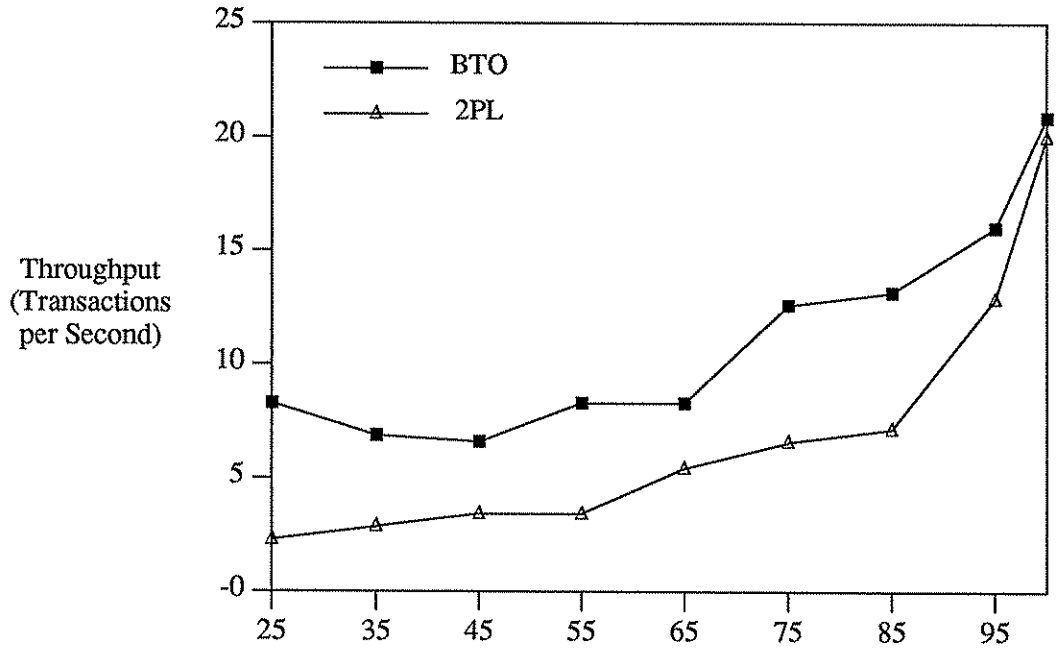

Figure 9. Throughput sensitivity to percent of transactions that are read-only

response time.

Another reason why BTO's bias against reads does not play a significant role in this experiment is that the bias is against long reads in the face of *short* updates. Here the updates are twice as long as those of the previous experiment. As a result, the "window of vulnerability" is much narrower for the reads in this experiment. In Figure 10, it is shown that since T_2's execution sequence is twice longer than in Figure 6, there is less opportunity for T_1 to read x in the interval where it would be forced to abort. Thus fewer aborts result in the BTO case than in Experiment 2, and BTO's response time is accordingly lower.

The other interesting facet of BTO's curve in Figure 8 is that it increases as reads increase from 25% to 50% of the transaction mix, and then steadily declines. The reason for this is the strictness requirement. From the time an update transaction writes to an object until it terminates, any other transaction accessing the object will be delayed so as to avoid the "cascading aborts" problem that may result from accessing uncommitted data objects. The probability of such a delay is maximized when the product

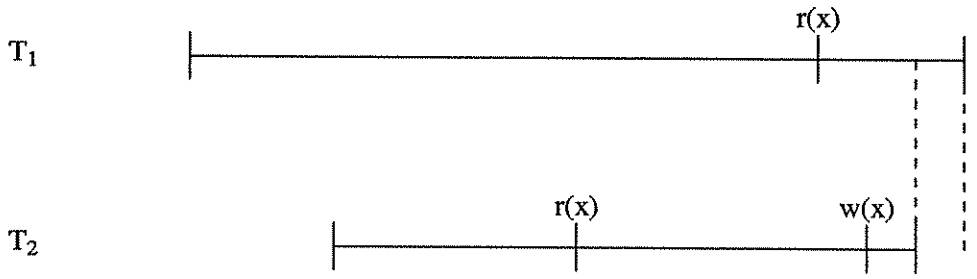

Figure 10. BTO reduced bias against long reads with long updates

of two probabilities is greatest: the probability of an update writing to an object, and the probability of another transaction conflicting with the update in that data object. This occurs in the midrange of the transaction mix; therefore the BTO curve peaks between 45% and 55% of reads. However, the BTO curve is not symmetrical about the 50% mark, since some updates will delay others to insure strictness, but no reads will delay other reads.

4.5. Analysis of Results

We have presented three experiments dealing with various aspects of distributed transaction processing. In Experiment 1, it was found that in case of low probability of conflicts, communication overhead will predominate the performance.

In Experiment 2, we have found the bias of BTO against long read-only transactions. This bias was explained in terms of the "window of vulnerability". This window extends from the end of a short update until the end of a long read. If a read-only transaction with a smaller timestamp accesses any of the update's data objects in the window of vulnerability, it will be forced to abort. It was seen that the abort process and restart delay become the dominant factors in determining performance. It was also found that 2PL's window of vulnerability, in which a deadlock would result, extends only from the time an update obtains a write-lock until the time it commits and the lock is released. When the updates are short, this window is much narrower than the analogous window for BTO, resulting in fewer aborts.

The results obtained in this experiment confirms those obtained in [Car86]. The difference in performance between BTO and 2PL is more pronounced in this experiment since the cost of aborting in a distributed environment is greater than in a centralized environment used in their study.

In Experiment 3, we have investigated the effect of varying the percentage of the read-only transaction in the workload on system performance. When updates are common, deadlock is more likely in the 2PL case, and thus predominates the performance. In BTO, two concurrent updates not necessarily conflict, and by using to the Thomas Write Rule, may not result in an abort. Therefore, the overhead of abort is less pronounced in BTO. In addition, 2PL is biased against updates that are considerably shorter than read-only transactions in the system. It is because updates are blocked, on the average, half the response time of the read-only transactions with which they conflict. It is also shown that BTO is biased against *short* updates, and that the window of vulnerability of long reads in the face of moderately long updates is much less pronounced, leading to fewer aborts.

In addition, we have seen the effect of strictness on performance; when updates are large and dominates the workload, they tend to delay all other transactions with which they conflict from the time they access their data objects until they commit.

In general, 2PL seems to suffer in a distributed environment from the requirement that each access, even a read, requires two messages. In addition, the overhead associated with any blocking will be magnified in a distributed system, since every block requires an additional message to notify the blocked transaction that the object is available. This is apparent not only in 2PL but also in BTO, since the strictness requires blocking all access requests until an update commits. Another disadvantage of 2PL in a distributed environment is the necessity of a deadlock detection. It requires wait-for information to be transferred among nodes, which is expensive in a distributed environment.

It was found that the transaction mix plays an important role in determining the system performance, since aborts are expensive in a distributed environment. If the transactions are mostly reads with moderately long updates, BTO will produce fewer aborts. However, BTO with strict execution requires

that all access to an object be blocked from the time of an update access until it commits. This blocking period will be more frequent and longer with more and larger updates. If the mix is mostly long reads with short updates, 2PL will produce fewer aborts, and the added expense of two messages per read access can be compensated by the lower abort overhead.

5. Conclusions

Prototyping large software systems is not a new approach. However, methodologies for developing a prototyping environment for distributed database systems have not been investigated in depth in spite of its potential benefits. In this paper, we have described the design and implementation of a prototyping environment that effectively models the salient features of a distributed system. There are three main goals to be achieved in developing a prototyping environment: modularity, flexibility, and extensibility. Modularity enables the environment to be easily reconfigured, since any subset of the available modules can be combined to produce a new testing environment. For instance, if a researcher wants to investigate the efficiency of various file servers in a distributed environment, he can use only those modules of the prototyping environment pertaining to message passing, server set-up, and client creation, and quickly develops a prototyping environment for his research with very little effort.

An additional benefit of the "right" modularity is that actual system software can be developed in the prototyping environment and then ported to the target machine. This is enabled by the use of technology-independent interfaces which are general enough to support any target system architecture. In addition to the portability, prototyped software may be run in a "hybrid" mode, that is, not all service calls need be prototyped. For example, file system calls in the prototype software can be intercepted by the interpreter and directed to the existing host file system. Then, as a prototype file system is developed, the file system calls can be directed to it. If the file system is not necessary or is not the focus of the current research, it need not be developed. This feature of the prototyping environment allows the developer to focus on only pertinent design issues.

Flexibility enables the environment to be applicable over a wide range of configurations and system parameters. One of the keys to achieving this goal is to design interfaces whose operations are independent both of the implementation technology and the context in which they are used. For example, the user-level Send operation sends an array of bytes to an abstract data type, the PortTag. Thus this operation can be used to send any packet type to any destination, be it local or distant.

The third goal is that the environment be extensible enough to model additional features of particular systems by adding modules without affecting the operation of or requiring the recompilation of existing modules. For instance, the implementation can be extended to model the operation of different types of I/O devices of different speeds by modifying the implementation module that performs the read and write operations. One way to modify the implementation would be to delay for a period depending on the address passed to the read or write operation. Reading from a disk might be indicated by one range of addresses and take some time, while reading from a tape drive might be indicated by another range and presumably take longer. However, because the interface of this module is device-independent, changing the implementation to process I/O requests at different speed will not affect any of the modules that request I/O operations. Therefore, time and effort for system reconfiguration can be reduced.

Expressive power and performance evaluation capability of the prototyping environment has been demonstrated by implementing a distributed database system and investigating its performance characteristics using two different concurrency control algorithms. We have shown that the transaction mix and the relative size of read/write sets of transactions are two important parameters that determine the system performance. Other important algorithms for database systems, including checkpointing, multiversion control, and priority-based scheduling, is under investigation using the prototyping environment.

References

[Agr85] R. Agrawal, M. Carey, and M. Livny "Models for studying concurrency control performance: Alternatives and implications," *Proc. ACM SIGMOD International Conference on*

Management of Data, Austin, TX, May 1985.

[Bern84] P. Bernstein and N. Goodman, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems*, December 1984, pp.596-615.

[Bern87] Bernstein, P., V. Hadzilacos, and N. Goodman, *Concurrency Control and recovery in Database Systems*, Addison Wesley, 1987.

[Boks87] C. Boksenbaum, M. Cart, J. Ferrie, and J. Pons, "Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, April 1987.

[Car84] M. Carey and M. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proc. 10th International Conference on Very Large Data Bases*, Singapore, Aug 1984.

[Car86] M. Carey and W. Muhanna, "The Performance of Multiversion Concurrency Control Algorithms," *ACM Transactions on Computer Systems*, November 1986, pp. 338-378.

[Cook87] R. Cook and S. Son, "The StarLite Project," *Fourth IEEE Workshop on Real-Time Operating Systems*, Cambridge, Massachusetts, July 1987, 139-141.

[Kohl86] W. Kohler and B. Jenq, "Performance Evaluation of Integrated Concurrency Control and Recovery Algorithms Using a Distributed Transaction Processing Testbed," *Proc. 6th International Conference on Distributed Computing Systems*, Cambridge, MA, May 1986, pp. 130-139.

[Ske81] D. Skeen, "Nonblocking Commit Protocols," *ACM SIGMOD International Conference on Management of Data*, Ann Arbor, 1981

[Son87] S. Son, "Synchronization of Replicated Data in Distributed Systems," *Information Systems 12*, 2, June 1987, 191-202.

[Son88] S. Son, "An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions," *Fourth International Conference on Data Engineering*, Los Angeles, Feb. 1988, 528-535.

[Son88b] S. Son, "A Message-Based Approach to Distributed Database Prototyping," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, 71-74.

[Son89] Son, S. H., "Checkpointing and Recovery in Distributed Database Systems," *Data Engineering 12*, 1, March 1989, 44-50.

[Tan85] A. Tanenbaum and R. Renesse, "Distributed Operating Systems," *ACM Computing Surveys*, December 1985, pp. 419-470.

[Wolf87] Wolfson, O., "The Overhead of Locking (and Commit) Protocols in Distributed Databases," *ACM Trans. Database Systems 12*, 3, Sept. 1987, 453-471.