A PROTOTYPING ENVIRONMENT FOR DISTRIBUTED DATABASE SYSTEMS

Sang H. Son
Yumi Kim

Computer Science Report No. TR-88-20
August 11, 1988

A Prototyping Environment for Distributed Database Systems

Sang H. Son
Yumi Kim

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

This paper describes 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 interprocess communication. The paper proposes the *port* construct to represent a flexible and modular message-communication facility. A general blocking construct is used for process scheduling and message-communication in simulated time. Based on these two notions, the paper describes the prototyping environment that has been developed and a series of experimentation performed for performance evaluation of a multiversion database system. 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 databases, prototyping, synchronization, transaction, communication, port

---

This work was supported in part by the Office of Naval Research under contract number N00014-86-K-0245, by the Department of Energy under contract number KC-07-01-02, and by the Federal Systems Division of IBM Corporation under University Agreement WF-159679.

1. Introduction

Performance evaluation of database systems has been carried out by several researchers using mostly simulation and analytic models [Agr85, Abb88, Liu87, Sing86]. However, there are important issues that have received little consideration in previous studies:

- Most of the previous modeling studies considered only the performance of synchronization algorithms in a single processor system. Few studies have addressed distributed environments.
- Since distributed algorithms must work correctly even with the subsystem failures, performance and reliability characteristics of database algorithms in degraded circumstances need to be studied.
- Although message-based simulations appear to be more natural for simulating distributed systems, a message-based approach to discrete-event simulation of distributed systems has not been fully developed.

Partly due to the reasons above, the field of distributed database system evaluation is currently in a state of disarray. Performance results are inconclusive and sometimes even contradictory [Wolf87]. We feel that an important reason for this situation is that many interrelated factors affecting performance (concurrency control, buffering schemes, data distribution, etc.) have been studied as a whole, without completely understanding the overhead imposed by each. An evaluation based on a combination of performance characterization and modeling is necessary in order to understand the impact of database control algorithms on the performance of distributed database systems.

Traditionally, database systems research has been concentrated in relatively specific areas such as file access methodologies, database language design, concurrency control and recovery, and query optimization. Surprisingly seldom, these separate components have been integrated in a single complete database system in a university environment. The reason is very simple: it is just too time consuming. By the time a complete system has been implemented, and is ready for performance evaluation, many of its component methodologies may have been superceded.

A prototyping technique can be applied effectively to the evaluation of control mechanisms 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. Especially if the system designer must deal with message-passing protocols and timing constraints, it is essential to have an appropriate prototyping environment for success in the design and analysis tasks.

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 [Son86, Son87, Son88], experimentation to verify their reliability properties and to evaluate their performance has not been performed due to the lack of appropriate test tools.

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

When prototyping distributed database systems, there are two possible approaches: sequential programming and distributed programming based on message-passing. Message-based simulations, in which events are message-communications, do not provide additional expressive power over standard simulation languages; message-passing can be simulated in many discrete-event simulation languages including SIMSCRIPT [Kiv69] and GPSS [Sch74]. However, a message-based simulation can be used as an effective tool for developing a distributed system because the simulation "looks" like a distributed program, while a simulation program written in a traditional simulation language is inherently a sequential program. Furthermore, if a simulation program is developed in a systematic way such that the principles of modularity and information hiding are observed, most of the simulation code can be used in the actual system, resulting in a reduced cost for system development and evaluation.

The rest of the paper is organized as follows. Section 2 presents an informal description of a message-based simulation. Section 3 discusses the *port* construct developed for inter-process communication in our prototyping environment. Section 4 describes the design principles and the current implementation of the prototyping environment. Section 5 presents a distributed database system with multiversion data objects, and its performance evaluation using the prototyping environment. Section 6 is the conclusion.

2. Message-Based Simulation

For a message-based simulation, the process view of simulation has been adopted. A distributed system being simulated consists of a number of *processes* which interact with others at discrete instants of time. Processes are basic building blocks of a simulation program. 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 simulation program models the dynamic

behavior of processes, resources, and their interactions as they evolve in time. Each physical operation of the system is simulated by a process, and the process interactions are called *events*.

In the literature, the notion of a process has been given numerous definitions. The definition used in our model is much the same as that given in [Bri78]: A process is the execution of an interruptible sequential program and represents the unit of resource allocation, such as the allocation of CPU time, main memory and I/O devices. As an example, consider the simulation of transaction processing in a distributed system to determine the response time distribution of user transactions submitted to the system. In the actual system, users submit requests and wait for the response from the system. User requests wait in a queue to be served by the transaction manager. The transaction manager initializes the data structure and determines the sites at which a portion of the transaction should be executed. The transaction manager then requests I/O to read (or write) data from (or into) the disk. In the simulation, the disk, transaction manager, and users are simulated by processes. Examples of events in this system are a user submitting a request and a transaction completing its I/O activity.

In message-based simulations, an event is the communication of a message. In the above example, the event of a user submitting the request may be simulated by a message being sent by the *user* process to the *transaction manager* process. We use the client/server paradigm for process interaction in our model. 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 how 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.

Internal actions of a process, i.e., actions that do not involve interactions with other processes in the system, are modeled either by the passage of simulation time or by the execution of sequential statements

within the process. We use a simulator clock to represent the passage of time in a simulation. The simulator clock advances in discrete steps, where each step simulates the passage of time between two events in the system. In message-based simulations, a process may wait until some time t specified by the process for a specific task to be done (e.g., disk I/O). While it is waiting for a specific task to be completed, messages to be received by the process are enqueued in a message buffer associated with the process. A process waits for the passage of simulation time by executing a blocking statement, which has the form

hold *t*

where t represents an integer-valued expression that specifies the maximum time to which the process is blocked. The process ceases to wait when the simulator clock reaches the value represented by t. In our transaction processing example, we may assume that it takes 10 milliseconds to write a value of a data object into the disk. This activity can be modeled by

hold (10).

The effect of executing this statement would be to cause the process to wait for 10 units of simulation time to elapse. From the perspective of the transaction manager process, if the value of the simulation clock was T before executing this statement, it would be $T + 10$ after the execution. However, the **hold** primitive is not sufficient to represent an activity that takes an unspecifiable amount of time. In the previous example, when an access request to a data object causes a conflict, the transaction manager cannot predict the amount of time it has to wait before the access privilege is granted. To represent such an activity, we use the **BlockProcess** statement, which when executed by a process cause it to enter the blocked state; the process remains in the blocked state until it is explicitly activated by another process.

In a physical system, each process makes independent progress in time, and many processes execute in parallel. In its simulation, the multiple processes of a physical system must be executed simultaneously on one processor. This simultaneity is achieved by interleaving the execution of different processes and

executing them in a quasi-parallel fashion. Besides the **hold** primitive which allows a process to suspend its execution for a predetermined time duration, other scheduling primitive, **PauseProcess,** is provided to guarantee quasi-parallel processing and finite progress of all active processes.

In message-based simulations, the communication of a message may take zero units of simulation time. For instance, in our transaction execution example, we may assume that the time taken from sending a message until its reception by the destination process is insignificant; thus, in the simulation, the transmission time for the message that models this event can be zero. However, nonzero transmission delays exist in distributed systems, and they can be modeled by causing the process sending (or receiving) a message to wait for a certain time corresponding to the message-transmission time before (or after) sending (or receiving) the message.

3. Inter-Process Communication and Ports

Processes communicate with each other by exchanging messages. This communication may be direct or indirect. In the case of direct communication, the sender of a message must specify the identifier of the destination process explicitly, i.e., to be able to communicate, the sending process must know the identifier of the destination process. For example, CSP [Hoa78] is based on direct communication. Direct communication is easy to understand and easy to use. Unfortunately, this kind of communication is inadequate for a number of communication patterns occurring in distributed database systems. If, for example, a server consists of several processes, each of which implements the same service, a client should be able to send a request which can be received by any process of the server. Using only a direct communication concept would be inconvenient because the client has to specify explicitly the destination process. Thus, a direct communication is unsuitable to support multi-process servers.

In case of indirect communication, processes indirectly communicate through message receptacles, which are called *ports*. Processes send messages to ports and receive messages by removing them from ports. Associated with each port is a queue, on which messages sent to this port and not yet removed by the associated process reside. It is possible that multiple processes share one port for receiving messages,

and several processes may send messages to the same port. In contrast to direct communication, a sender does not identify a destination process but a port, which may be shared by a set of destination processes. Therefore, in indirect communication, a sender does not have to know the destination process. If indirect communication is used, the multi-process server problem can be solved very easily. A service is logically associated with a port, and each client that requires the service sends its request to this port. The server processes providing the service receive the requests from this port.

An assumption implied by indirect communication as described above is that a request directed to a multi-process server can be executed by any process of that server. In general, this assumption is not valid in the context of distributed database systems, where the requests of a transaction often have to be performed by a particular process in a server. As discussed below, we solve this problem of indirect communication by using two different bindings between ports and server processes.

Clients and servers communicate with one another by exchanging *request* and *answer* messages. A client requests a service by sending a request message to a server providing the desired service. Each request specifies a set of operations to be executed by the corresponding server. A server receiving a request message performs the request and, if necessary, sends an answer message back to the client. In our model, each request is associated with (or belongs to) a particular transaction, that is, a server receiving a request executes it as part of the transaction the request is associated with. Thus, a server process can be classified by the number of transactions for which it executes requests: a *single-transaction* server and a *multi-transaction* server. A single-transaction server p works on behalf of only one transaction, i.e., all requests p executes belongs to the same transaction, while a multi-transaction server q may work on behalf of more than one transaction.

In order to provide single-transaction servers, the system must support a dynamic process structure, that is, for each transaction requesting the service, an individual process is created. This process executes all requests from the transaction and then terminates. Consequently, for each transaction there exists exactly one server process, and each of the server processes works for only one transaction, while in case

of multi-transaction servers, the server must multiplex itself between transactions. Since requests are often context-sensitive in most database systems, the transaction-specific context information must be stored or supplied by the client together with each request message, resulting in an increase in complexity and overhead. However, in systems in which process creation is expensive, creating a process for each request might lead to a substantial overhead. We have implemented both types of server processes in the prototyping environment in order to compromise the complexity and overhead.

In port-oriented inter-process communication, ports are usually associated with services, where a client can ask for a specific service by sending a request message to a port associated with that service. Each port is owned by one or more processes, which are called the *owner process group* of the port. Only the processes in the owner process group may create and destroy the port and may remove messages from the port.

Port-oriented communication supports a one-level indirection: a client sends a request to a port, which is mapped to the set of server processes that are able to remove the request from it. This mapping is invisible for the client; it is only known by the processes in the owner group, which can manipulate the mapping. A owner process group may possess more than one ports associated with the service in order to support the concept of single-transaction servers. In that case, each port is associated with a specific transaction. Thus, a owner process group may possess the following two types of ports:

- *Specific port:*

  Each specific port is associated with a specific transaction. The specific port of a transaction receives request messages of this transaction for the service. A service is associated with at most one specific port per transaction.

- *Generic port:*

  Each service is associated with at most one generic port. The generic port of a service receives request messages from any transaction, for which no specific port exists for the service. The generic port of a service is shared by all processes in the owner process group of the service.

Initially, each service is associated with a generic port. If the service is supported by a multi-transaction server (e.g., lock managing service), it is associated with only one generic port. All the request messages are sent to this port, and performed by the server process associated with the port. If the server consists of several processes, any of them can remove request messages from the associated generic port and execute the requests. For example, the system may provide two independent generic ports for lock management: *access-request* port and *release-request* port. All the requests for data access come to the access-request port, and all the unlock requests come to the release-request port.

An alternative to this multi-port approach is a single-port approach in which a server is associated with only one generic port. For example, lock managing service is associated with the *lock-management* port, and all the lock-related requests (access request, release request, status checking request, etc.) come to this port. In this case, each message contains the *type tag* which specifies the type of the message.

For a single-transaction server, the system must be able to create a new process as well as a specific port associated with the process. When the first request of a transaction t arrives at the generic port of the service, a new specific port associated with t is created, and the request is placed into it. Moreover, a new process, $p(t)$, which is the owner process of the new port, is created. All the succeeding requests for t are placed into this port, and neither a new process nor a new port is created when any of these requests arrives. Process $p(t)$ has exclusive access to all requests from t, because it is the only owner process of the port. When the transaction t is terminated, $p(t)$ as well as the associated port can be destroyed. Consequently, each process created in this way works for one transaction, and all requests of a given transaction are performed by a single process. For example, consider the transaction management in a distributed system. Transaction management service is initially associated with a well-known generic port. When a new user transaction is submitted into the system, a transaction manager process is created with a specific port associated with it. After that, all the inter-process communication of the transaction is performed by that process using the associated port. When the transaction is terminated, the transaction manager process and the associated port will be destroyed.

4. Structure and User Interface of the Prototyping Environment

For developing high performance distributed database systems, design of a specific control mechanism or evaluation of implementation alternatives is not sufficient. Control mechanisms need to be integrated with the operating system for high performance. This is 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. There are several projects investigating methodologies for this integration of database functions with operating systems [Hask87, Spec87]. 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/reliability characteristics for 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. Since one of the design goals of the prototyping environment is to conduct experiments to provide empirical evaluation of the design and implementation of database algorithms for synchronization, communication software, and transaction processing support in operating systems, it has built-in support for performance measurement of both elapsed time and blocked time for each transaction.

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, written in Modula-2. The StarLite kernel supports process control to create, ready, block, and terminate processes. It also supports the Semaphore type, which is used by higher-level modules in resource control, critical section implementation, and synchronous message-passing. Scheduler in the kernel maintains the simulation clock and provides the **hold** primitive to simulate the passage of time.

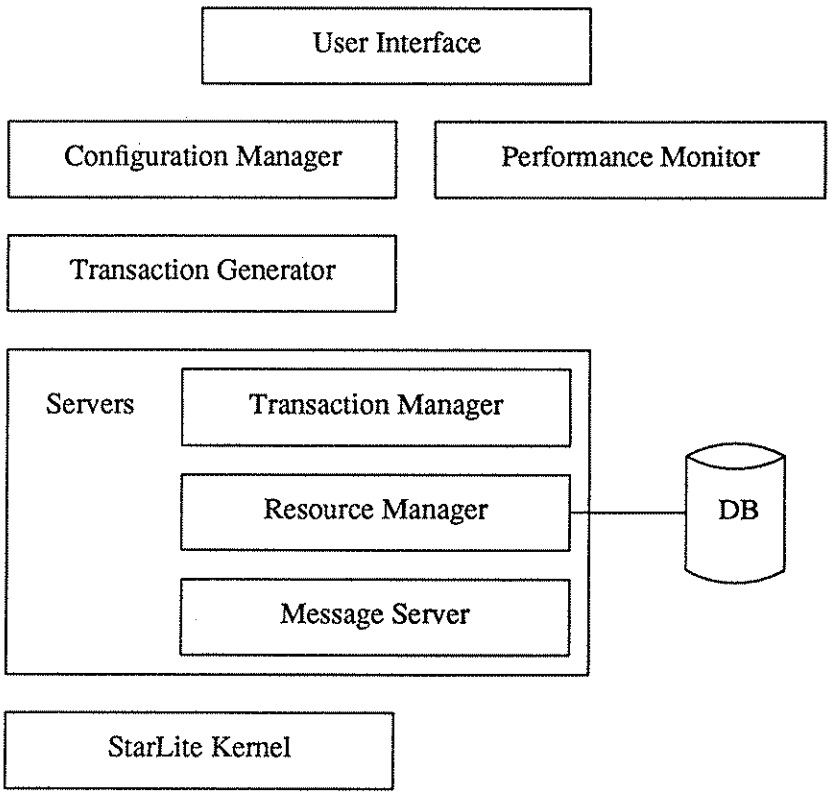


Fig. 1. Structure of the prototyping environment

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.
- database configuration: database at each site with user defined structure, size, granularity, and levels of replication.
- 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.
- failure characteristics: the site and the time of a crash, and the type of recovery to be performed.

UI initiates the Configuration Manager (CM) which initialize 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. If the access request cannot be granted, Transaction Manager (TM) executes either blocking operation to wait until the data object can be accessed, or aborting procedure, depending on the situation. 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.

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, 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. If the sender wishes to block itself to wait for a response, it can wait on a semaphore by performing the **WaitforResponse** primitive. In this way, 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.

5. Prototyping A Multiversion Database

The previous section described the structure and implementation of the prototyping environment. In this section, we present a multiversion 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 a distributed database system, and 2) performance comparison between a multiversion database system and its corresponding single-version database system through the sensitivity study of key parameters that affect performance. The key parameters evaluated include set size, transaction ratio, interarrival time, and the database size.

5.1. Multiversion Control

In a multiversion database system, each data object consists of a number of consecutive versions. The objective of using multiple versions is to increase the degree of concurrency and to reduce the possibility of rejecting user requests by providing a succession of views of data objects. One of the reasons for rejecting a user request is that its operations cannot be serviced by the system. For example, a read operation has to be rejected if the value of data object it was supposed to read has already been overwritten by some other user request. Such rejections can be avoided by keeping old versions of each data object so that an appropriate old value can be given to a tardy read operation. In a system with multiple versions of data, each write operation on a data object produces a new version instead of overwriting it. Hence, for each read operation, the system selects an appropriate version to read, enjoying the flexibility in

controlling the order of read and write operations.

The multiversion database system we have implemented is based on timestamp ordering. Each data object is represented as a list of versions connected by a doubly linked pointer, and each version is associated with timestamps for its creation and the latest read, and a valid bit to specify whether the version is certified. The data structure for multi-version control is shown in Figure 2. R_ts and W_ts are initialized to 0. Tag bit is 0 for uncertified versions, and 1 for certified versions.

5.2. Transaction Execution Using Multiversion Data Objects

Each transaction consists of read and write requests for data objects. Read requests are never rejected in a multi-version database system if all the versions are retained. We discuss the issue of version maintenance in the next section. A read operation does not necessarily read the latest committed version of a data object. A read request is transformed to a version-read operation by selecting an appropriate version to read. The timestamp of a read request is compared with W_ts of the highest available version. When a read request with timestamp T is sent to the Resource Manager, the version of a data object with the largest timestamp less than T is selected as the value to be returned. Figure 3 shows an example of a read operation with timestamp "11".

The timestamp of a write request is compared with the read timestamp of the highest version of the data object. A new version with the timestamp greater than the R_ts of the highest certified version is

```
ListPointer = POINTER TO data_object;

data_object = RECORD
                R_ts : TimeStamp;       (* read timestamp *)
                W_ts : TimeStamp;       (* write timestamp *)
                Tag : CARDINAL;         (* Tag bit for validation check *)
                Value : Integer;        (* data value *)
                Up, Down : ListPointer; (* pointer to upper or lower version *)
              END;
```

Fig. 2. Data structure of a data object.

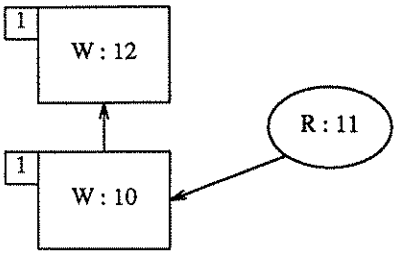


Fig. 3. A read operation with two certified versions of a data object.

built on the upper level, with a "0" tag bit to indicate that the new version is not valid yet.

Figure 4 shows the creation of a new version as a result of a write request with timestamp "12" for a data object dset[0]. The system first checks the timestamp of the highest version of dset[0] (in this example, the R_ts of the version is "10"). Since the R_ts is less than the timestamp of the write request, the system can create a new version on upper level with "0" tag bit. This new temporary version is not valid until the transaction that created the version commits and changes the tag bit from "0" to "1". It is possible that a read request reaches the Resource Manager later than the write request with larger timestamp. For example, R_ts of dset[0] was "0" when the next version was created, and then a read request with timestamp "10" has arrived. The system can process this retarded read request by returning the value of the version with W_ts of "0", and update its R_ts to "10".

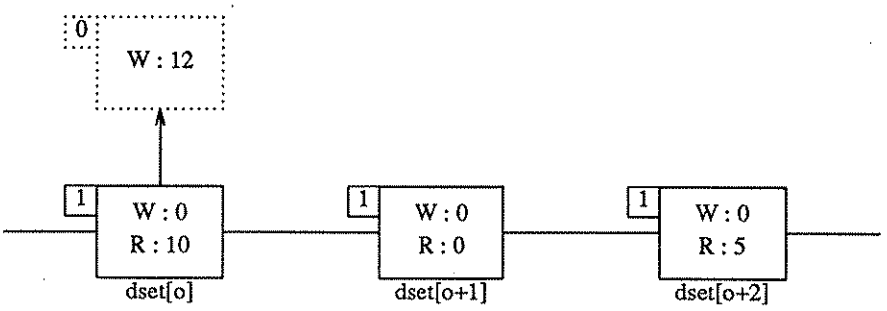


Fig. 4. Creation of a new version for data object dset[0].

In order to simplify the concurrency control mechanism, we allow only one temporary version for each data object. Inserting a new version in the middle of existing valid versions is not allowed. Figure 5 shows two scenarios for write requests. The write request with the timestamp "22" is aborted because there already exists a temporary version for this data object. Another write request with the timestamp "8" is also aborted because it will create a version to be inserted in the middle of the existing versions.

5.3. Version Retention

All versions cannot be retained forever in any multiversion database systems due to the storage and processing overhead. We now discuss how versions can be discarded when they are not needed by read-only transactions. Recall that each data object keeps track of the read requests that have accessed the data object. By using time-stamp assignment method for read-only transactions different from that for update transactions, version management can be simplified. When a read-only transaction begins, the coordinator

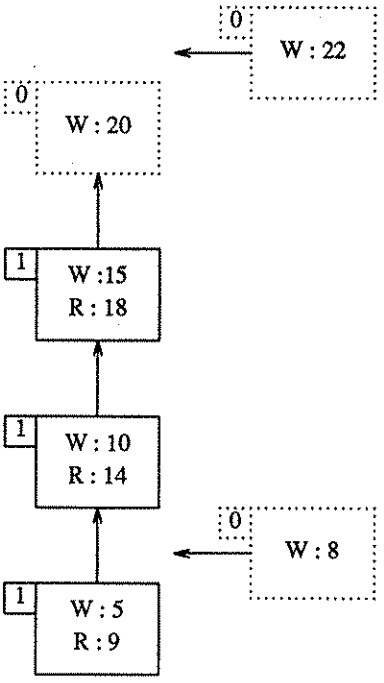


Fig. 5. Rules for version creation

sends messages to the participants telling them the data objects the transaction needs to read. When a participant receives such a request, it checks the current time-stamp of each data object at the site, and sends the maximum time-stamps among them to the coordinator. Each data object accessed by a read-only transaction in this way records the pair of the identifier of that transaction and the current time-stamp it reported. After receiving responses from all participants, the coordinator chooses a unique time-stamp greater than all the responses. The time-stamp recorded for the read-only transaction at each object is thus a lower bound on the time-stamp of the transaction, and it will be used in making a decision to discard or retain versions of the data object.

When a read-only transaction with time-stamp TS invokes a read operation on a data object, the participant chooses the version of the data object with the largest time-stamp less than TS. This invocation of read operation is nothing but sending the time-stamp TS to the participants, since each participant already knows which data object to read. If TS is larger than the current time-stamp of the data object, it will be updated as TS. This will force update transactions that commit later to choose time-stamps larger than TS, ensuring that the version selected for the read-only transaction does not change. Resource Manager can use the following rule to decide which versions to keep and which to discard.

Rule for retention:

A version with the write time-stamp TS must be retained if

(1) there is no version with time-stamp greater than TS (i.e., current version), or

(2) there is a version with time-stamp TS' > TS, and there is an active read-only transaction whose time-stamp might be between TS and TS'.

By having a read-only transaction inform data objects when it completes, versions of data objects that are no longer needed can be discarded. This process of informing data objects that a read-only transaction has completed need not be performed synchronously with the commit of the transaction. It imposes some overhead on the system, but the overhead can be reduced by piggybacking information on existing mes-

sages, or by sending messages when the system load is low.

When a read-only transaction sends a read request to an object, Resource Manager effectively agrees to retain the current version and any later versions, until it knows which of those versions is needed by the read-only transaction. When Resource Manager finds out the time-stamp chosen by the transaction, it can tell exactly which version the transaction needs to read. At that point any versions that were retained only because the read-only transaction might have needed them can be discarded. By minimizing the time during which only a lower bound on the transaction's time-stamp is known, the system can reduce the storage needed for maintaining versions. One simple way of doing this is to have each read-only transaction broadcast its time-stamp to all Resource Manager when it chooses the time-stamp.

The version management described above is effective at minimizing the amount of storage needed for multiple versions. For example, unlike the "version pool" scheme in [Chan85], it is not necessary to discard a version that is needed by an active read-only transaction because the buffer space is being used by a version that no transaction wants to read. However, ensuring that each Resource Manager knows which versions are needed at any point in time has an associated cost; a read-only transaction cannot begin execution until it has chosen a time-stamp, a process that requires communicating with all data objects it needs to access.

5.4. Performance Evaluation

The experiment was conducted to measure the average response time and the number of aborts for a group of transactions running on a multiversion database system and its corresponding single-version system, both implemented using the prototyping environment. Two groups of transactions with different characteristics (e.g., type and number of access to data objects) were executed concurrently. The objective was to study sensitivity of key parameters on those two performance measures. The details of this study are given in [Kim88]. Here we present our findings briefly.

Performance is highly dependent on the set size of transactions. As shown in Figure 6, a multiversion database system outperforms the corresponding single-version system for the type of workload under

which they are expected to be beneficial: a mix of small update transactions and larger read-only transactions. The reason for this is that, in a multiversion database system, a read requests have higher priority than the write requests, while such priority for read requests is not provided in a single-version system. Therefore, in a single-version system, the probability of rejecting a read request is equal to that of a write request. This result holds for a variety of transaction mix. As shown in Figure 7, a single-version database system outperforms its multiversion counterpart for a different transaction mix.

As shown in Figure 8, it was observed that the performance of a multiversion system in terms of the number of aborts is better than its single-version counterpart for a mix of small update transactions and larger read-only transactions. Similar experiments have been performed by changing the database size and the mean interarrival time of transactions. It was found, however, that the main result remains the same. From these experiments, it becomes clear that among the four variables we studied, the size of transaction read-set and write-set is the most sensitive parameter for determining the performance of a multiversion database system.

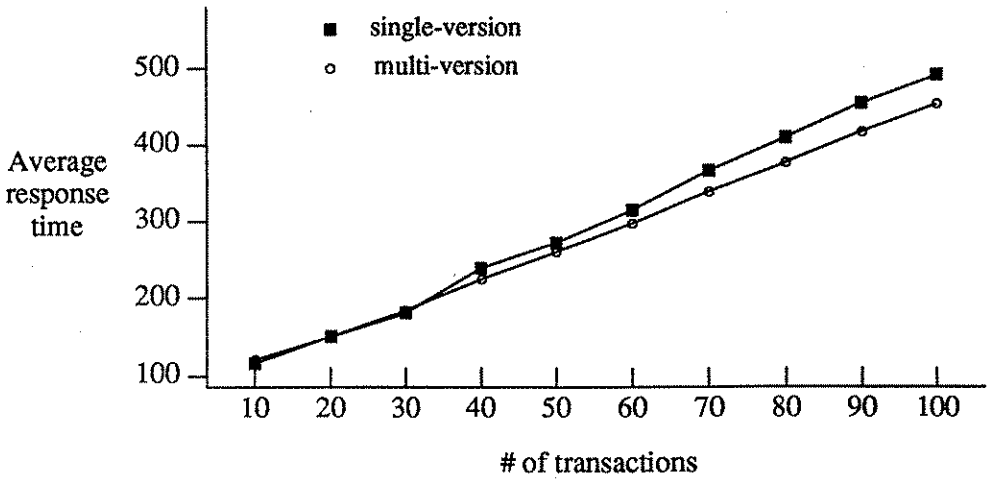
6. Conclusions

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 presented a prototyping environment that has been developed based on the StarLite concurrent programming kernel and message-based approach with modular building blocks. Although the complexity of a distributed database system makes prototyping difficult, the implementation has proven satisfactory for experimentation of design choices, different database control techniques, and even an integrated evaluation of database systems. It supports a very flexible user interface to allow a wide range of system configurations and workload characteristics.

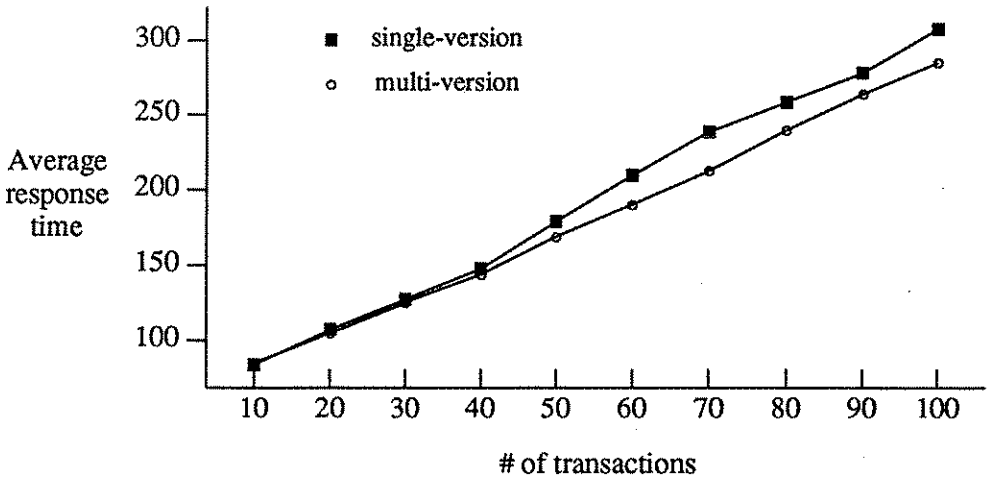
To develop an effective prototyping environment for distributed database systems, appropriate operating system support is essential. Since our prototyping environment is designed to provide a spectrum of database functions and operating system modules, it facilitates the construction of multiple

system instances with different characteristics without much overhead. Expressive power and performance evaluation capability of our prototyping environment has been demonstrated by developing a multiversion distributed database system and its single-version counterpart, using the prototyping environment. We have shown that a multiversion database system outperforms its single-version counterpart when the workload has desirable characteristics, and that set-size of transactions is one of the most sensitive parameters affecting the performance.

The current implementation of the prototyping environment also provide multi-transaction execution facility based on two-phase locking. Using this facility, we plan to investigate priority-based concurrency control algorithms and associated priority inversion problem. *Priority inversion* is said to occur when a higher priority process is forced to wait for the execution of a lower priority process. In order to maintain a high degree of schedulability, priority inversion must be minimized. One approach to this problem has been proposed based on the notion of *priority inheritance* [Sha88]. As part of a joint effort with researchers at Carnegie-Mellon University and IBM Federal Systems Division, the database prototyping environment presented in this paper will be used for the evaluation and comparison of the priority inheritance protocol with other design alternatives. If successful, the prototyping environment will demonstrate the feasibility of sharing software and experimental results with researchers at different institutions.

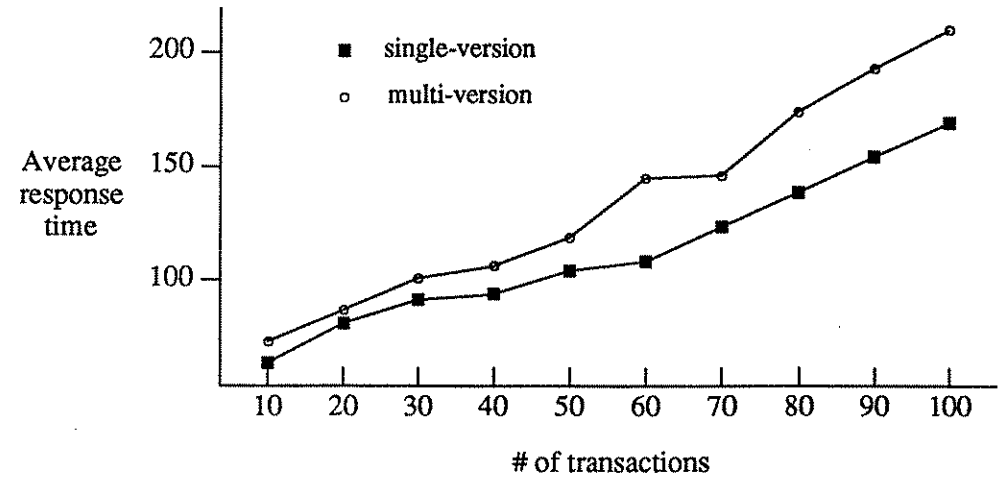


PARAMETERS
Group 1 : Setsize = 10, Type = READ-only, Transaction Ratio = 80%
Group 2 : Setsize = 2, Type = WRITE-only, Transaction Ratio = 20%

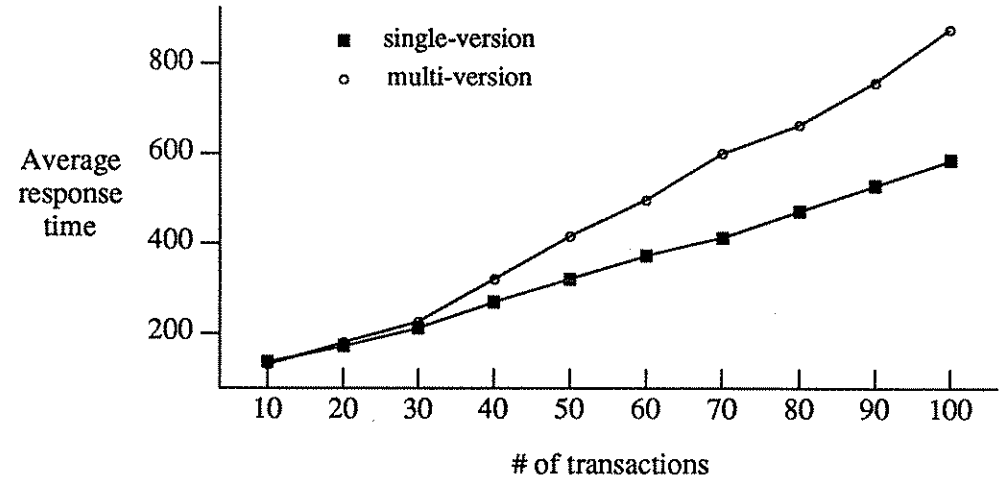


PARAMETERS
Group 1 : Setsize = 10, Type = READ-only, Transaction Ratio = 50%
Group 2 : Setsize = 2, Type = WRITE-only, Transaction Ratio = 50%

Fig. 6. Average transaction response time

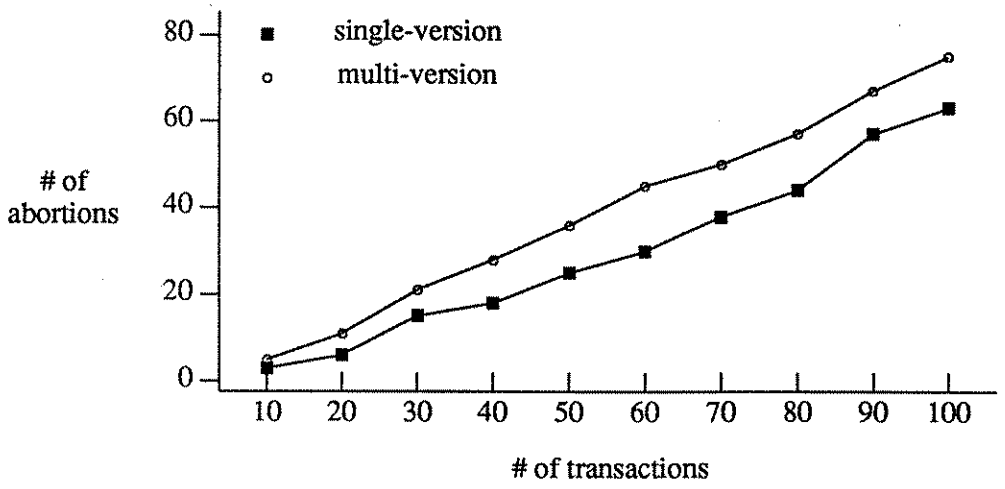


PARAMETERS
Group 1 : Setsize = 2, Type = READ-only, Transaction Ratio = 80%
Group 2 : Setsize = 10, Type = WRITE-only, Transaction Ratio = 20%

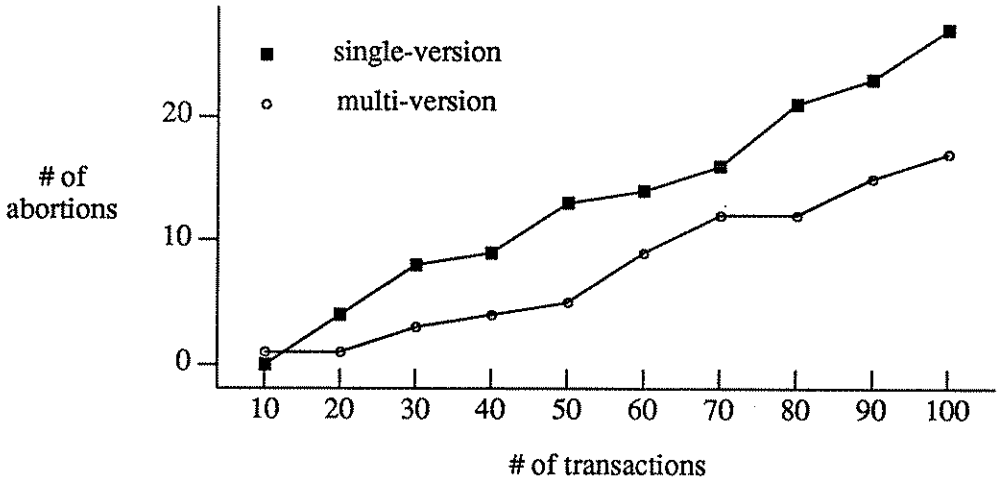


PARAMETERS
Group1 : Setsize = 2, Type = READ-only, Transaction Ratio = 20%
Group2 : Setsize = 10, Type = WRITE-only, Transaction Ratio = 80%

Fig. 7. Average transaction response time



PARAMETERS
Group 1 : Setsize = 2, Type = READ-only, Transaction Ratio = 20%
Group 2 : Setsize = 10, Type = WRITE-only, Transaction Ratio = 80%



PARAMETERS
Group 1 : Setsize = 10, Type = READ-only, Transaction Ratio = 20%
Group 2 : Setsize = 2, Type = WRITE-only, Transaction Ratio = 80%

Fig. 8. Number of abortions

References

[Abb88] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Study," *Tech. Rep. CS-TR-146-88*, Dept. of Computer Science, Princeton University, Feb. 1988, (to appear in VLDB Conference, Sept. 1988).

[Agr85] Agrawal, R., M. Carey, and M. Livny, "Models for Studying Concurrency Control Performance: Alternatives and Implications," *ACM SIGMOD Conference*, 1985, 108-121.

[Bri78] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *Comm. of the ACM 21*, 11, Nov. 1978.

[Chan85] Chan, A. and R. Gray, "Implementing Distributed Read-Only Transactions," *IEEE Trans. on Software Engineering SE-11*, 2, Feb. 1985, 205-212.

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

[Hask87] Haskin, R. et al., "Recovery Management in QuickSilver," *11th ACM Symposium on Operating Systems Principles*, Austin, Texas, Nov. 1987, 107-108.

[Hoa78] Hoare, C. A. R., "Communicating Sequential Processes," *Comm. of the ACM 21*, 8, Aug. 1978, 666-677.

[Kim88] Kim, Y., "Performance Evaluation of Multiversion Data Objects in Distributed Database Systems," *Tech. Rep.* Dept. of Computer Science, University of Virginia, June 1988.

[Kiv69] Kiviat, P., R. Villareau, and H. Markowitz, *The SIMSCRIPT II Programming Language*, Englewood Cliffs, NJ, Prentice-Hall, 1969.

[Liu87] Liu, J. W. S., K. J. Lin, and S. Natarajan, "Scheduling Real-Time, Periodic Jobs Using Imprecise Results," *Real-Time Systems Symposium*, Dec. 1987, 252-260.

[Sch74] Schriber, T., *Simulation Using GPSS*, NY, Wiley, 1974.

[Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 82-98.

[Sing86] Singhal, M. and A. Agrawala, "A Concurrency Control Algorithm and Its Performance for Replicated Database Systems," *6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, May 1986, 140-147.

[Spec87] A. Spector, et al., "Camelot-An Interim Report," *Tech. Rep. CMU-CS-87-129*, Carnegie-Mellon University, June 1987.

[Son86] Son, S. H. and A. Agrawala, "An Algorithm for Database Reconstruction in Distributed Environments," *6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, May 1986, 532-539.

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

[Son88] Son, S. H., "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] Son, S. H., "A Message-Based Approach to Distributed Database Prototyping," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, 71-74.

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