M.I.T. Laboratory for Computer Science
Computer Systems Research Division

February 22, 1978
Request for Comments No. 158

A Message Based Model of a Distributed Data Base System

By Warren Montgomery

The problem of maintaining the consistency of a shared data base that is being manipulated by asynchronous, independent, transactions has been extensively studied [1,2,3,4]. Most of this research has made the assumption that the data base consists of passive objects which are shared by the processes performing the transactions. These processes can access the objects in any order. Although the communication network used in a distributed system may constrain the order in which accesses made by different processes will be performed, the constraints were not used in these studies. This research has developed elaborate locking strategies to insure that concurrent updating does not result in inconsistency.

This report takes an alternative view in which each data object in the data base is accessible to only a single process, and processes communicate by sending and receiving messages, possibly consisting of the data objects themselves. In this way, the communication paths are made explicit. The meaning of the notion of consistency in this system is explored, and strategies for performing concurrent transactions while maintaining consistency are developed. As an example, a simple distributed banking system is presented and implemented as a consistent message-based system.

I) Data Base Consistency.

A data base consists of a set of <u>items</u> representing a collection of information. In general, the information in a data base is redundant such that one can make assertions about the items in a data base that should always be true, due to this redundancy. An example of such an assertion is that the sum of all of the assets of a corporation is always equal to the sum of all liabilities. A data base is said to be <u>consistent</u> if it satisfies all such assertions.

This note is an informal working paper of the M.I.T. Laboratory for Computer Science, Computer Systems Research Division. It should not be reproduced without the author's permission, and it should not be cited in other publications.

The operations to be performed on data bases are known as __transactions__. Each transaction obtains the values of a set of input items, and updates a set of output items. A transaction that updates several different items may temporarily put the data base in an inconsistent state before all of those updates are completed. If the consistency assertions are to be maintained, then such inconsistent states must not be visible to other transactions on the data base. This effect can be achieved by forcing the transactions to be __atomic__, in that either all or none of the updates performed by a particular transactions are reflected in the values of the data items seen by other transactions.

In order to insure that transactions are atomic, a locking protocol must be enforced so that a process performing a transaction does not see the inconsistent states that may occur during the execution of other transactions. Several papers [1,2,3], discuss the problem of controlling the concurrent execution of transactions so that each sees a consistent version of the data base.

Gray et. al. [1] give definitions for four different levels of consistency and discuss locking strategies to achieve each. The notion of consistency used in this paper corresponds to their level 3, which is the most difficult to achieve (in terms of the restrictions needed on concurrent execution), and appears to be desirable for most applications.[1] The sequencing constraints required to achieve this level of consistence are presented in several forms. In one form, they can be stated as a requirement that the ordering <<< on the transactions, where T_1 <<< T_2 if T_1 accesses some object A before T_2 access A, and either T_1 or T_2 modifies A, be extendible (by transitive closure) to a partial order. Thus the result of running some set of transactions concurrently is always the same as that of running the same set sequentially, in some order consistent with the <<< ordering.

---

1. While the authors claim that forcing all transactions to see level 0 or level 1 consistency allows transactions to be constructed to see higher levels of consistency, and may save locking overhead by allowing many transactions to run at the lower levels of consistency, they also point that output values produced by a transaction reflect the level of consistency that that transaction saw. These low-level consistency values are propagated by __any__ transaction that reads them, so that transactions desiring a high level of consistency can never read values produced by those observing a lower level. Thus low level of consistency transactions would appear to have very limited use.

The locking strategies developed are efficient, in that they allow the data base to be constructed so that a high degree of concurrency may be obtained with little locking overhead. The extension of these strategies to a distributed database, however, is not clear.

A second paper [2] gives a general discussion of the problem of controlling concurrent transactions. This paper gives a model for distributed data bases in which the data is partitioned among sites and each transaction is performed by a process that migrates among the sites that hold the values that it accesses. Each site is responsible for controlling the execution of transactions at that site, and the sites communicate only when a transaction is moved and when a transaction is completed. The authors describe a class of control algorithms that work by assigning an order to the transactions to be processed and use that order to resolve conflicts between processes attempting to access the same data, possibly by aborting and restarting them. The necessity of restarting some transaction that has completed a substantial amount of processing is undesirable, but seems unavoidable in this model of concurrency control.

A third paper[3], gives a method of analyzing the set of transactions to be performed on a data base to determine the amount of locking needed. The data base is assumed to be replicated at several sites, each of which performs atomic transactions on its local copy and distributes the resulting updates as atomic actions to be performed by other sites. The fact that each transaction is performed atomically at some site using copies of the data items available at that site, and the knowledge of the complete set of transactions that are performed by the entire system are used to greatly reduce the locking required to insure consistency. Although the system is claimed to be applicable to data bases in which each site contains only a partial copy of the data base, a transaction that requires access to a set of items for which there is no single site that has copies of all of them appears to cause difficulties.

These papers have all been based on a model of a data base in which the data objects are shared by the processes representing the transactions. The fact that two of these transactions may share concurrent access to objects that represent the same data items leads to the requirement that the transactions be ordered (by locking) so that each sees a consistent version of the data base.

Modeling the data base system as a message-based system, in which each object is accessible to only one process, and objects are exchanged by message passing, may be much more desirable for a distributed data base system.

First, a process in a distributed system can only reference directly objects that are maintained locally. If two processes at different sites are to share an item, then this sharing must be implemented by some form of message passing. The messages that must be passed between processes in order to implement sharing could be combined with those needed to synchronize references, thereby reducing the number or size of messages needed. A common pattern of sharing is one in which a great many processes reference an object, but do so very infrequently. Thus even if updates are much less frequent than references that are not updates, several updates may occur between two references by a particular process. Such a pattern of sharing can be most efficiently implemented by supplying new values of the object to a process making infrequent references at each reference, not at each update.

Although the models discussed so far consider the communication between sites to be unconstrained, many communication networks now built or under development constrain inter-site communication and impose some order on many inter-site messages. Nearly all communication networks enforce the constraint that messages flowing between two sites are totally ordered, such that messages arrive and are processed in the order in which they were sent.

In addition, some network topologies allow a message to be broadcast to a set of sites as an atomic transmission, such that the reception of a broadcast message by all destinations is consistently ordered with the reception of other messages sent to those destinations. A model of a distributed data base system which includes these constraints may allow substantially simpler and more efficient synchronization of transactions.

II) The Meaning of Consistency in a Message-Based System.

If a data base is expressed as a message based system, in which the objects that represent data items are accessible to only one process at a time, then the requirements for consistency are somewhat different. We need not require that the consistency assertions always hold on all of the items, wherever they may be, because no transaction or outside observer can obtain the value of an item without sending messages to a process in possession of an object that represents that item, and receiving a value as a reply. We need only require that the view of the data base seen by processes carrying out transactions (which consists of responses to requests for values of items), be consistent. Before going into the details of consistency in such a system, I will present a sample message-based system as a framework for the discussion.

What I will present here is a method of describing a distributed application as a collection of processes communicating only through message passing. Each process has a number of ports through which it can send or receive messages to or from the ports of other processes. Ports are typed in that they send or receive messages consisting of objects of some abstract data type. Ports are also designated as either input ports or output ports depending on whether they send or receive messages.

A distributed application is implemented as a network of such processes constructed connecting each output port of each process to some input port of some process. Several outputs can be connected to a single input, and the streams of values produced by those outputs are merged. Each input port has a queue of messages to be processed. Ports that are connected must match in type. If the application has external input or produces external results, some of the input or output ports of the processes are left unconnected and are designated to receive or produce these external inputs or results.

Each process performs atomic processing steps known as **atomic process steps**. [1] Each such

1. Atomic process steps are the basic unit of computation which can be performed atomically with respect to other process steps taking place in the system. Several such atomic steps may be needed to implement a single transaction.

step consumes some set of inputs to that process and produces some set of outputs. The process steps that can be performed by a single process are individually specified as consuming values from input ports to that process and producing results at output ports. Each process may perform a number of different kinds of atomic steps. All process steps performed by a single process share the objects local to that process and thus the results of an atomic step can depend on both the input values to that step and on the steps previously performed by that process (which may have had side effects on the local variables of that process).

This descriptive system is similar to the actors formalism of Hewitt [5]. The important distinctions between this model and actors are the presence of multiple, typed input ports to each process (actor) and the ability to specify certain atomic steps which consume messages from several ports. These were included because they appear to be common needs in the construction of distributed systems and simplify the task of describing applications. The specification of a process is very similar to the behavioral specifications used by Hewitt, in that each kind of process step can be viewed as a different behavior for a process (selected by the messages received at that step), and each step can be specified by the messages that it produces and the change in the internal state of the process.

We can now use this system to describe a distributed data base. Initially, I assume that the data is partitioned among a collection of sites, and that each item is represented by a single object appearing at a single site. Duplication of data for reliability could be handled at a lower level than the level of description used here, using an algorithm like that developed by Alsberg [6], or Thomas [7]. A later section of this report considers a more general scheme for incorporating duplicated data into this model.

Three kinds of processes will be used to describe the data base system:

> Data managers, each of which maintains some portion of the data base as objects in its internal state.

Transaction processes, which formulate requests to the data managers in order to carry out transactions.

Communication processes, which convey messages between the transaction processes and the data managers.

The requirement for data base consistency expressed above can be restated in terms of ordering the atomic process steps of the data managers that carry out parts of transactions. Each transaction causes a set of atomic steps to be performed by the manager processes for the items accessed by that transaction. Each of the steps performed by the managers is performed for some transaction. Each data manager M_i views the transactions as occurring in some order defined by the order in which the atomic steps for those transactions occurred in M_i. Thus for each manager, we have an order $<_i$ on the transactions, such that $T_1 <_i T_2$ iff both T_1 and T_2 include process steps of M_i and a process step related to T_1 preceded a process step for T_2, and one of these two steps caused a change in the internal state of M_i. The data base system remains consistent if the transitive closure of the union of all $<_i$ is a partial order on the transactions.

The communications processes will constrain the concurrent execution of transactions so as to insure that this partial order is consistent by constraining the order in which messages are delivered to the managers, and thus the order of execution of the process steps related to individual transactions. It is in general hard to insure consistency without global knowledge of the transactions in progress, however we will show that for some classes of transactions a simple interconnection of the communication processes, and simple protocols can be used to insure consistency.

In order to study the internal structure of a transaction, we need a model for what the transaction does. For this purpose, I adopt a model similar to the L-U graph of SDD-1[3] which I will call an <u>activity graph</u>. A transaction can be viewed as a mapping from a set of input items to a set of output items. In this mapping, not all of the outputs depend on each input. The activity graph of a transaction is a directed graph whose nodes are the data items read or written by that transaction, and with directed arc between each input item, and each output that depends on that input.

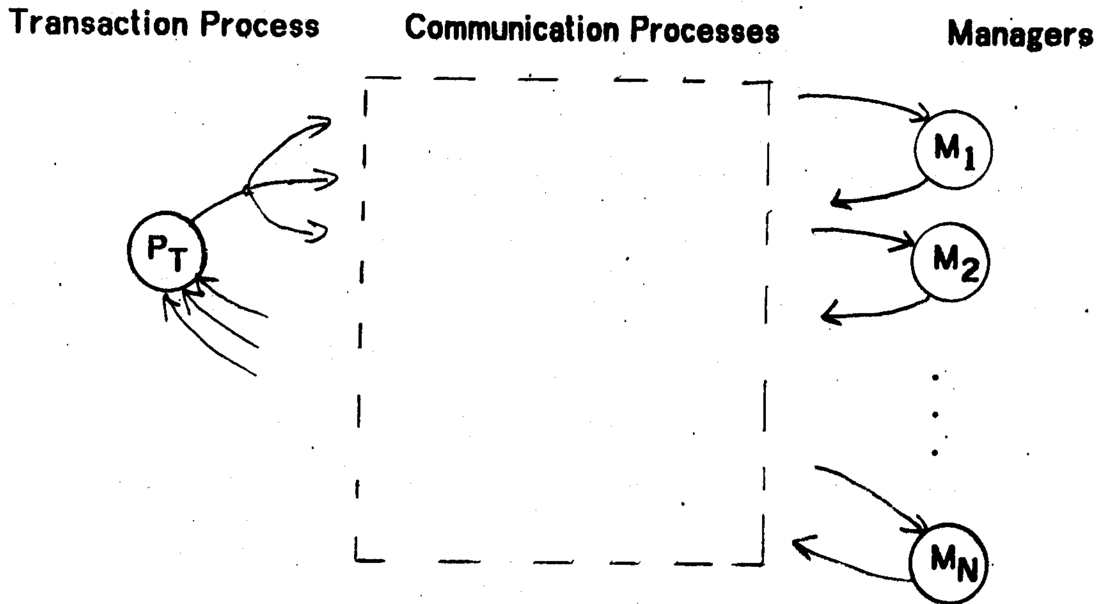
One common class of transactions for which the coordination problem is simplified are those which to not have cross-manager dependencies. That is, the accesses to the data base made by each manager in performing such a transaction do not depend on data obtained from other managers. Such transactions can be recognized by the fact that their activity graphs have no arcs leading from an item represented by an object controlled by one manager, to an item represented by an object controlled by another manager.

This kind of transaction can be implemented using only one atomic step of each manager, in which the part of the transaction pertaining to items local to that manager is performed. The transaction process for this kind of transaction has one atomic step that emits a set of requests for the managers. If the transaction returns a value, the requests to the managers cause values to be sent back to the transaction process, and a second atomic step of the transaction process uses these values to obtain the returned value of the transaction. Figure 1 illustrates the process structure for this kind of transaction. We have not yet considered the structure of the communication processes that route the requests to the managers. Figure 2 shows an execution of such a transaction. The figure shows the atomic steps of the transaction process and the manager processes, with a directed arc between a step that produces a message and one that consumes it. The steps have three part names consisting of the name of the transaction, the name of the process and the particular kind of process step invoked. (Recall that processes can have several different kinds of steps.)

This class of transaction includes all "retrieval" transactions that do not modify the data base. The activity graph of a retrieval transaction contains no arcs, as no items are modified.

The ordering requirement for a set of transactions with no cross-manager dependencies is that the ordering on the transactions perceived by each manager (based on the order in which it receives its requests) is mutually consistent with those perceived by all other managers. The requests sent by a transaction process are sent as an atomic transmission. The atomic transmissions are partially ordered, such that two transmissions that include requests sent to the same manager are ordered. The order in which requests are delivered to the managers is consistent with the order of the atomic transmissions that sent those requests.

Figure 1
A Transaction With No Cross-Manager Dependencies

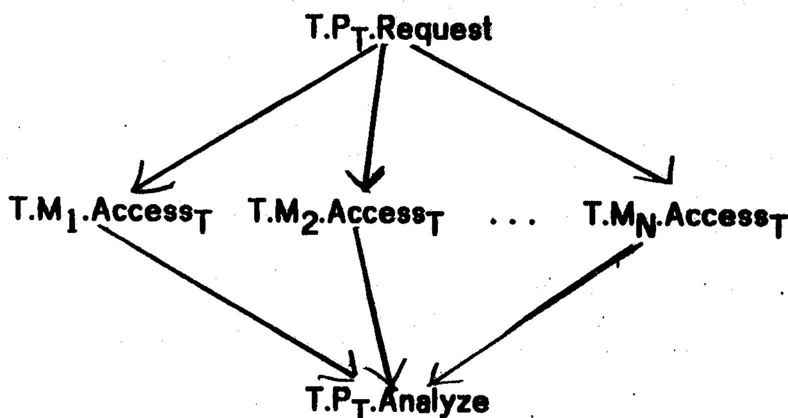


One way in which the ordering constraints needed for atomic transmissions can be enforced is through the use of timestamps [7]. Each transaction process could send one message, containing all requests, to a communications process. This communication process could assign a unique timestamp to the transaction, and associate that timestamp with all of the requests. For each manager, we could have a communication process that received the requests for that manager and passed them on in timestamp order. The protocol needed by the communication processes to enforce proper ordering is complicated and expensive.

Some communication networks enforce the needed synchronization easily. An ether net [8] allows a message to be broadcast to a set of receivers as one atomic transmission. An ether net being used to connect the transaction processes to the managers can be modeled as a single communication process, which receives a message containing all of a transaction's requests from a transaction process, and distributes these requests to the managers in one step. A ring network

Figure 2

An Execution of a Transaction with No Cross-Manager Dependencies



imposes a similar constraint.[1] These communications strategies can be simulated in software, if not present in the communication hardware.

Using only one communication process to route requests to the managers may overly restrict concurrency if the communication medium is not sequential. Rather than use a single communication process, we can use a hierarchy of communications processes to distribute the requests. In this case, a transaction process sends a message containing its requests to some communication process that is above (in the hierarchy) all of the managers that are destinations of those requests. On receiving a message containing a collection of requests, a communication process determines which requests should be sent to each of its children in the hierarchy in order for each request to reach its eventual destination. Each request is forwarded to the correct child, again packaging all requests going to the

1. A common problem with broadcast transmissions is that some recipients may miss a broadcast, because of lack of buffering. This could be remedied by sequencing all broadcast transmissions, having several sites keep a log of recent broadcasts, detecting lost broadcasts and having a node request a lost broadcast from one of the logs, and preventing a node from issuing new broadcasts until it has heard all old ones. The details of the implementation of reliable broadcasts are still under investigation.

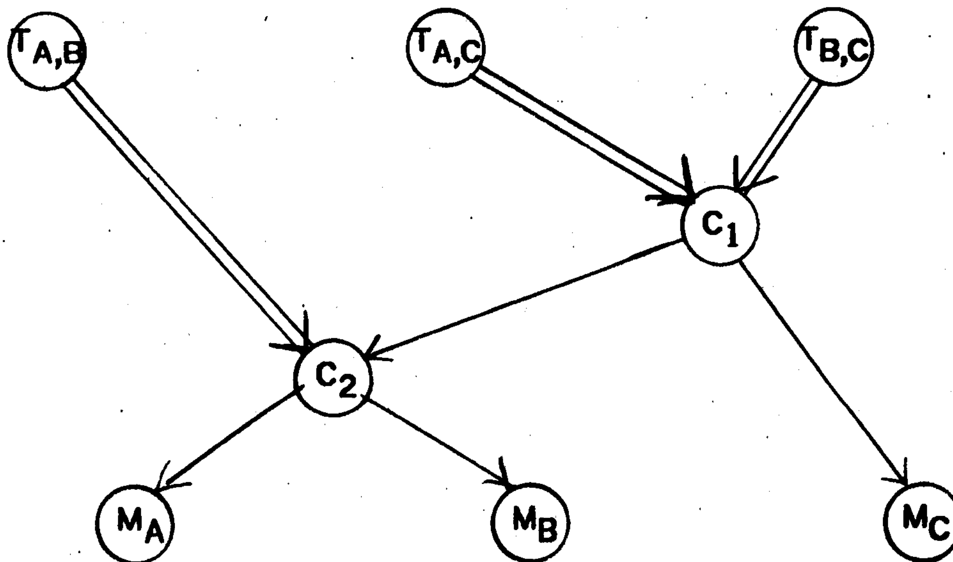
same child in a single message.

Two transactions that reference overlapping sets of items are sequenced by the communication process that is above all of the items referenced by both transactions. This sequencing insures that all of the requests of a transaction are delivered to the managers in as if they were a single atomic transmission.

Figure 3 gives an illustration of such a hierarchy. There are three manager processes, and two communication processes in the hierarchy. Three transaction processes are shown, each of which sends its requests to the communication process that is above all of the needed managers in the hierarchy.

Figure 3

A Hierarchical Communication Network



The class of transactions discussed so far require no distributed locking in that no transaction is prohibited from using or updating the data base while computation is being performed at some remote site.[1] Transactions performing updates that depend on the value of data items represented by objects under the control of several different managers do inherently require some form of locking. These updates cannot be computed before the inputs have been obtained. The items being updated must effectively be locked at the same time that the input items are obtained, so that accesses to the updated items made by other transactions do not occur before the updates are made. This locking can be described in this system by a transaction process that executes a pair of steps, one which emits messages that request the values of all inputs to the transaction being implemented and lock all output items, and one which receives the requested inputs and produces the eventual outputs. This locking prevents any other accesses to those items until the second step of the transaction process produces new values for the updated items.

The three different types of requests (perform local access, lock item, and update), require different treatment by the communication processes in order to minimize the restriction of concurrency by locking. An update message should be delivered to the appropriate manager as quickly as possible, as the lock that it will release may be holding up other transactions. Local access requests should also be delivered promptly, as they may be holding up a transaction that also holds a lock.

Sending a lock request for an item is not the same as locking that item. A lock request must be sent in the same atomic transmission with other requests for values or local accesses needed to complete the transaction. The communication processes, however, operate asynchronously and could choose to deliver the lock request last. If the transaction does not require the previous value of the item being locked, then the new value for that item can be computed before the lock request is delivered. This allows transactions that accesses the item which is the target of the lock request, but do not otherwise interfere with the locking transaction to perform accesses to that item while the

1. The sequential processing of messages by the managers may restrict concurrency. This restriction can be avoided by assigning fewer data objects to each manager and using more manager processes.

locking transaction is executing, and they appear as if they occurred before the locking transaction.

As an example, consider the transactions and hierarchy of Figure 3. Let $T_{A,C}$ be C = f(A), where f is a complicated function requiring a great deal of computation. $T_{A,C}$ must send a message to C_1 requesting the value of C and a lock on A. C_1 must deliver the value request to M_C before the transaction can continue. C_1 could, however, follow a "lazy" forwarding scheme for lock requests, such that a lock request is not forwarded to the next manager or communication process until some other request for the same destination has been produced. This would cause the lock request to be held by C_1 until some other request for C_2 is produced. A transaction $T_{A,B}$ involving only A and B could be performed concurrently with the computation of f(C) by $T_{A,C}$. When this value is computed and sent by $T_{A,C}$, the update request "forces" the lock request through the hierarchy.

III) An Example, A Distributed Bank Account System.

As an example of a consistent distributed data base system, consider a distributed data base containing all of the accounts belonging to a single bank (including both customer accounts and internal accounts representing such quantities as the cash in tellers' drawers or loans from the Federal Reserve). The data base is partitioned with a part located at each branch of the bank, and the only consistency assertion that must be maintained is that the sum of all of the balances at all of the branches must always be zero. We wish to provide for two kinds of transactions, one which moves data from one account to another, and one which obtains a set of balances and performs some function on them, returning the result to the user. An example of such a transaction is one that sums some set of balances.

Both of these two kinds of transactions have no cross-manager dependencies. The value obtaining transaction does not modify the data base, and so it can be implemented by a transaction process with two steps, one of which requests the values of the desired items, and another that applies a function to the requested values. The money moving transactions also have no cross manager dependencies, as the new value for each of the balances touched depends only on the

previous value and the amount moved.[1]

The value-obtaining transactions could in practice take a long time to complete. Consider, for example a transaction that obtained all balances and summed them in order to check consistency. It would be desirable if such a long running transaction did not unnecessarily delay other transactions, yet proceeded steadily, without being aborted to allow other transactions to run. The hierarchical model described in this memo provides an effective solution to this problem.

If the manager processes each have control of a relatively small part of the data base, then the effort to obtain all values under the control of any one manager is relatively small. A particular transaction is only delayed until the requests previously sent to the managers that it accesses have been processed. Thus only a small part of a long running value obtaining transaction must be performed in order to allow later transactions that access the same values to proceed. In a system with a limited number of real processors being multiplexed among the manager processes, a priority scheme could be adopted to give priority to managers with requests pending from high-priority transactions.

IV) Possible Extensions and Directions for Further Work.

The method of describing distributed database systems presented here is not quite as general as that of Stearns [2]. The model presented here requires that the set of objects that a transaction will read or write be known in advance, before it has accessed any data values. While it is anticipated that most transactions are of this form, and other researchers have made the same assumption [3], a method of dealing with transactions that must view some data values before knowing the set that it

---

1. This example is perhaps oversimplified, as most banks also require that balances of savings and checking accounts do not become negative. If the money-moving transaction is constructed to enforce this constraint, a cross-manager dependency is introduced because new value of the credited balance depends on the old value of the debited balance. In practice, a loose consistency constraint requiring each transfer from a customer account to be preceded by a request for the balance on that account, and limiting the amount that can be moved in one transfer is generally sufficient. This could be done entirely by the transaction process submitting the money moving transactions, thus giving the user the illusion that the movement takes place in one step. A manager that discovers transaction that takes too much out of an account could also initiate a "backup" transaction to undo the offending transaction.

updates is needed.

If the uncertainty about the input or output sets of such a transaction is small, it could be modeled by requesting values or locks for all possible inputs and outputs. A more general scheme would be needed for a transaction with greater uncertainty.

A second unanswered question about this system is its reliability. The reliability question is considered in greater detail in a companion report. Techniques have been developed to allow information to be recorded safely, such that it is not lost during a crash. Communication protocols can be constructed to reliably deliver a stream of messages from one process to another. One fundamental problem that cannot effectively be solved is allowing a manager to release a lock after a failure has occurred, and not cause inconsistency. By reducing the need for locking in this system, we have reduced the chance that a failure can occur with a lock set. The other report gives a strategy for dealing with failures that occur with locks set.

Another related area for investigation is the ability to change the communication paths used, by changing the interconnection of communication processes, to avoid a failed site. Such changes must be done carefully, so that the requests held by the failed site are not lost, and are properly sequenced with other requests.

This report presents a model for distributed data base systems in which the data base is not redundantly stored. The extension of this model to a redundantly stored data base appears to be straightforward, and simpler than some others. Suppose that some of the data items are represented by duplicated objects, with the copies under control of different managers. Thus in addition to synchronizing the transactions, we must insure that the copies remain identical. This can be done by requiring that any transaction that modifies a value must send requests to all managers that hold copies of that value. Any copy of a duplicated value may be used as input to a transaction.

In order to modify a redundantly stored item, a transaction may need to send requests to many managers. In the hierarchical communication system, this requires that such a transaction send its requests to a communication process high up in the hierarchy. Thus updating redundantly stored data is expensive, both because of the number of messages that must be sent, and because it may require additional synchronization. Accessing a redundantly stored object, however, is equivalent to accessing one of the copies. If a number of copies are available, the copy used by a particular transaction can be chosen to minimize the communication and synchronization costs.

The performance of a data base system built on this model depends heavily on the assignment of data objects to managers and the communication network. If few, large managers are used, concurrency is reduced, but few cross-manager dependencies occur, requiring little locking. If many small managers are used, the amount of locking needed increases, but so does the potential for concurrency. Finding a good partition the data base for a particular set of transactions is an interesting research problem.

V) Conclusions

Viewing a distributed system as a collection of processes communicating through message passing appears to be useful way to gain insights into the efficient structuring of applications. During the next few months, I intend to continue to investigate the unanswered questions in the previous section in order to arrive at general algorithms for structuring distributed applications. Any comments on these ideas or their extensions would be greatly appreciated.

References

[1] Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L., "Granularity of Locks and Degrees of Consistency in a Shared Data Base", IBM Research Report RJ 1654, September, 1975

[2] Stearns, R., et al., "Concurrency control for database systems," IEEE Symposium of Foundations of Computer Science CH1133-8 C, October, 1976, pp. 19-32.

[3] Rothnie, J.B., Bernstein, P.A., Goodman, N., and Papadimitriou, C.A., "The Redundant Update Methodology of SDD-1: A System for Distributed Databases," Computer Corporation of America Technical Report, February, 1977.

[4] Eswaran, Gray, Lorie, Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM 19, 11, November, 1976.

[5] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," M.I.T. Artificial Intelligence Laboratory, A.I. Memo #410, December, 1976.

[6] Alsberg, P.A., Belford, G.G., Day, J.D., Grapa, E., "Multi-Copy Resiliency Techniques," CAC Document # 202, May, 1976.

[7] Johnson, P.R. and R.H. Thomas, "The Maintenance of Duplicate Databases," ARPANET NWG/RFC #677, January 1975.

[8] Metcalfe, R.M., et al., "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM19, No. 7, pp. 395-404, July, 1976.