The Implementation of Reliable Distributed Multiprocess Systems*

Leslie Lamport **
Massachusetts Computer Associates, Inc., 26 Princess Street, Wakefield, Mass. 01880, USA

A method is described for implementing any system by a network of processes so it continues to function properly despite the failure or malfunction of individual processes and communication arcs; where "malfunction" means doing something incorrectly, and "failure" means doing nothing. The system is defined in terms of a sequential "user machine", and a precise correctness condition for the implementation of this user machine is given. An algorithm to implement the user machine in the absence of malfunctioning, and a rigorous proof of its correctness, are given for a network of three processes with perfect clocks. The generalization to an arbitrary network of processes with imperfect clocks is described. It is briefly indicated how malfunctioning can be handled by adding redundant checking to the implementation and including error detection and correction mechanisms in the user machine.

Keywords: Computer networks, distributed computing, reliable synchronization, system specification.

Leslie Lamport received a B.S. in mamathematics from M.I.T., and an M.A. and Ph.D in mathematics from Brandeis University. He has worked at the Mitre Corporation, Marlboro College and Massachusetts Computer Associates, and is currently at SRI International. He has done reserach in analytic partial differential equations, and in various aspects of concurrent processing. His current interests include the theory of concurrent processes, the foundations of quantum mechanics, and applied oriental philosophy.

* This research was supported by the Advanced Research Projects Agency of the Department of Defense and by Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F30602-76-C-0094.

** Author's current address: Computer Science Laboratory, SRI International, Menlo Park, California 94025, USA

Computer Networks 2 (1978) 95–114

1. Introduction

1.1. Goals

By a distributed multiprocess system, we mean a system composed of physically separated processes which communicate with one another by sending messages. We will describe an algorithm for implementing any system as a highly reliable distributed multiprocess system. The reliability of such a system involves two major goals.

Goal 1. To enable the system to continue functioning despite the failure of one or more processes or communication lines.

Goal 2. To enable the system to function correctly despite the malfunctioning of one or more processes or communication lines.

We say that a component "fails" when it completely stops functioning. We say that it "malfunctions" if it continues to operate, but performs one or more operations incorrectly. The two goals are therefore quite different. Our primary concern is Goal 1, we will only briefly discuss Goal 2.

To illustrate the objective of our algorithm, consider an airline reservation system. Such a system is usually implemented by using a single central computer to process the reservations. If that computer fails or loses communication with the reservation stations, then the system stops functioning. A special case of our algorithm allows the system to be implemented with three computers at separate locations. It will continue to process reservations normally so long as any two of the computers and the communication lines joining them are functioning properly. The general algorithm allows an implementation by an arbitrary network of computers, and enables the system to continue operating so long as a large enough portion of the network is functioning properly.

We make no assumptions about what happens when a process or communication line has failed. In particular, we do not assume that a process can detect that a communication line or another process has failed. A process need not even detect its own failure, but may at any time continue from the point in its algorithm at which it had failed.

It may be hard for the reader to appreciate the difficulty of this problem if he has not tried to solve it himself. For example, an obvious approach for an airline reservation system with three computers is to have each one vote on whether to grant a reservation, and to require two "yes" votes for it to be granted. However, suppose that three different requests for a seat on the same flight are issued concurrently at different sites, and there are only two seats left. Each computer will vote "yes" on two of the requests, so it is possible for all three reservations to receive two "yes" votes and thus be granted–thereby overfilling the flight. [1] We hope that this example gives some indication of how difficult the problem is.

What it means to achieve Goal 1 is a subtle question, and we know of no previous work which approaches it rigorously. Correctness proofs of algorithms usually ignore questions of physical execution time. However, the concept of failure is meaningless without a notion of physical time. We can only tell that a computer system has failed ("crashed") when we have been waiting too long for a response. The first part of this paper is devoted to defining exactly what Goal 1 means.

Our primary aim is to describe and prove the correctness of a general algorithm for achieving Goal 1. However, we are faced with a dilemma. The general algorithm is somewhat complicated, and a complete discussion of it would be quite long. Moreover, a rigorous analysis of any algorithm requires simplifying assumptions. Were we to restrict ourselves to such an analysis, we might give the impression that our algorithm is useless because it is based upon unrealistic assumptions. However, a thorough discussion of implementation details would be unbearably long. We have therefore chosen a compromise approach. We will give a rigorous exposition only for a special case of our algorithm, and will just sketch the general algorithm. We will also discuss how the algorithm can be used as the basis for a reliable distributed system, although many details will be omitted.

We wish to emphasize that we are concerned with a practical method for *implementing* a system which has already been specified. We will therefore discuss, at least briefly, the most difficult implementation problems. Space limitations have forced us to ignore many problems whose solutions we felt to be straightforward. It is inevitable that some things which seem straightforward to us will not be obvious to some readers. We can only assure the reader that we have tried to discover all the problems that would arise in an implementation, and have not knowingly hidden any that we could not solve. Only an actual implementation can determine if we have overlooked any difficult ones.

The remainder of Section 1 is devoted to a precise statement of Goals 1 and 2. In Section 2, we describe and prove the correctness of our algorithm for a particular network of processes containing idealized clocks. In Section 3, we indicate how the algorithm can be generalized to an arbitrary network of processes with real clocks. Section 4 discusses how our algorithm can be used as the basis for a practical total system which satisfies Goals 1 and 2. An index of symbols and special terms is included at the end of the paper.

1.2. Logical specification of the system

1.2.1. The user machine

In order to prove that a system is implemented correctly, we need a way of defining precisely what the system is supposed to do. We consider the operation of the system to consist of receiving *commands* and generating *responses*. For example, Table 1 gives some commands and possible responses that they might generate in an airline reservation system.

Let us suppose for now that all the commands are issued sequentially by a single user. We can then specify the system by a state machine, which we call the *user machine*. The user machine is defined by a set S

Table 1
Examples of commands and possible responses in an airline reservation system

Command	Response
1. Request 1 seat for Jones on Flight 221 for 2/7/80.	1. Flight filled, Jones placed on waiting list.
2. Cancel reservation for Smith on Flight 221 for 2/7/80.	2. (a) Smith's reservation cancelled, (b) Jones moved from waiting list onto Flight 221 for 2/7/80.
3. How many seats are left on Flight 123 for 2/7/80?	3. 27 seats left.

[1] We are considering an idealized airline reservation system in which flights may not be overfilled. Allowing a flight to be overfilled simplifies this particular problem, but does not lead to any useful general approach.

of possible states, a set C of possible commands, a set R of possible responses, and a mapping e: $C \times S \rightarrow R \times S$. The relation $e(C, S) = (R, S')$ means that executing the command C with the user machine in state S produces the response R and changes the user machine state to S'.

Specifying a system by a user machine is a conceptually simple task. For example, a state for an airline reservation system might consist of a set of flights, where each flight consists of a flight number, date, capacity, list of reservations granted, and waiting list. After specifying the sets of commands and responses, it is in principle easy to define the mapping e which specifies exactly what the airline reservation system is supposed to do. More precisely, defining e is a straightforward *sequential* programming problem.

As another example, we consider a distributed file system. Defining the individual read and write operations as user machine commands might be impractical, for reasons which will become clear later. Instead, the user machine could be employed only to acquire and release files. The actual reading and writing of the files would then be external to the system specified by the user machine. The user machine state would include a directory of the files, but not the contents of the files themselves. The actual specification of such a user machine is simple and of little interest.

As this example indicates, the user machine might specify a *synchronizing kernel* of a larger system. The relation of the user machine to the total system will be discussed in Section 4. Now, we are only concerned with implementing the system which is specified by a given user machine. The details of this user machine do not concern us. We need only observe that the user machine is deterministic and that its action is defined for every command, state pair. (An invalid command, such as requesting a seat on a nonexistent flight, can simply produce an error message response and leave the user machine state unchanged.)

1.2.2. Multiple users

So far, we have pretended that all commands come from a single user. We not consider the real case in which there are a number of users who can issue commands concurrently. For example, each reservation station of an airline reservation system might be a separate user. To employ the user machine, it is necessary to *sequence* the commands from the different users to form a single stream of commands, and to *distribute* the responses to the appropriate users. Logically, the system then appears as in fig. 1. We emphasize that this is a logical description only. Achieving Goal 1 requires that none of the boxes in fig. 1 be implemented by a single component. Each of the three logical functions – sequencer, user machine, and distributor – must be physically distributed throughout the system.

We have to specify what correctness conditions the sequencer and distributor must satisfy. We make the following obvious and simple requirement for the sequencer.

SC. If a command C_1 is issued before a command C_2, then C_1 must precede C_2 in the user machine's command sequence.

A discussion of what "before" means in the hypothesis is beyond the scope of this paper, and we must refer the reader to [1]. It is shown there that in some cases neither command is issued before the other, so condition SC does not completely specify the sequencing of commands in the command stream.

Correctness of the distributor simply means that responses are sent to the appropriate users. Note that different parts of a single response may have to be sent to different users. Implementing the distributor is a straightforward problem in message routing, and will not concern us. We will therefore ignore the distributor, except for a brief mention in Section 4.3.4 of its role in achieving Goal 2.

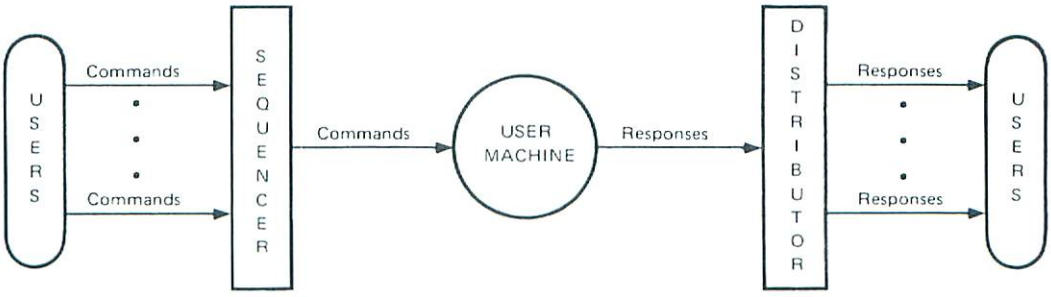

Fig. 1. The logical system.

1.2.3. Response time

There is one crucial element that has so far been left out of our discussion: time. Correctly executing the user machine is not enough; we must also require that a response be generated reasonably soon after a command is issued. An airline reservation system is of little use if it can take weeks to process a reservation. We would therefore like to require that there be some length of time Δ such that the response to a command is generated within Δ seconds after the command is issued. (For linguistic convenience, we use the word "second" to denote an arbitrary unit of time.)

Communication failure may make it impossible to execute a particular command within Δ seconds after it is issued. It turns out to be difficult for a system to execute such a command later without either delaying the execution of other commands or violating condition SC. We therefore introduce the possibility of rejecting such a command, so it does not appear in the user machine's command stream. Our logical system now appears as shown in fig. 2. The accepter decides whether a command is accepted as input to the user machine or is rejected. A command should normally be accepted, and it should be rejected only because of the failure of some physical components. A precise correctness condition for the accepter must depend upon the physical configuration of the system, and will be given in Section 1.4.

There should also be some provision for notifying a user if his command is rejected, but we will ignore this problem. Implementing such notification would require only a simple addition to our algorithm.

Having introduced the accepter, we can now require that the response to every accepted command be generated within Δ seconds after the command is issued, for some fixed parameter Δ. The value of Δ must depend upon the physical details of the system. For any general algorithm, one can only express Δ in terms of worst-case message transmission and response times.

The "functioning" of the system mentioned in Goal 1 can be defined by this requirement plus some condition on when commands must be accepted. Note that because we want to reglect the implementation of the distributor, we merely require that the response be *generated* within Δ seconds. We will not worry about when it reaches the user(s).

1.3. The physical system

Fig. 2 shows the logical specification of the system. We now describe the "physical" system of processes with which this logical system must be implemented. We assume a network composed of processes and one-way communication lines, such as the one shown in fig. 3. A process can communicate with a neighboring process by sending messages over a communication line. This might represent sending messages between two processes in a single computer via the operating system, or between two computers on different continents via satellite. The details of message transmission will not concern us.

In an airline reservation system, the bookkeepers would be the computers which process reservations. The users might be the reservation stations together with any other processes that can issue commands or receive responses; e.g., there might be a user process at every airport that must know how many meals to order for each flight.

1.4. Reliability

Goals 1 and 2 require that the system continue to operate correctly despite the failure or malfunction of some physical components. Equivalently, we require that the system continue to operate correctly so long as a large enough portion of the network of

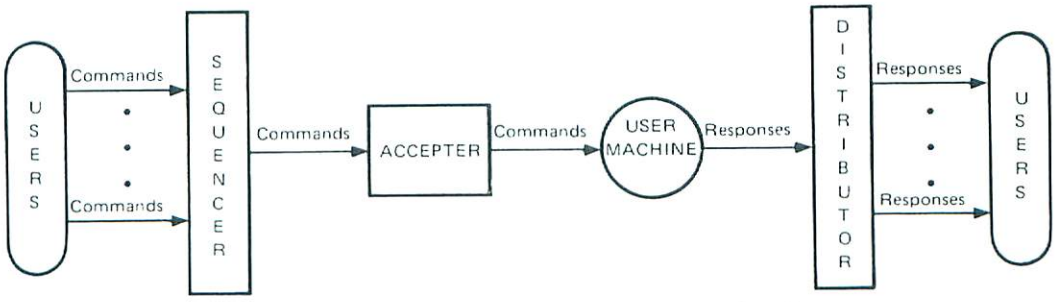

Fig. 2. Logical system when commands may be rejected.

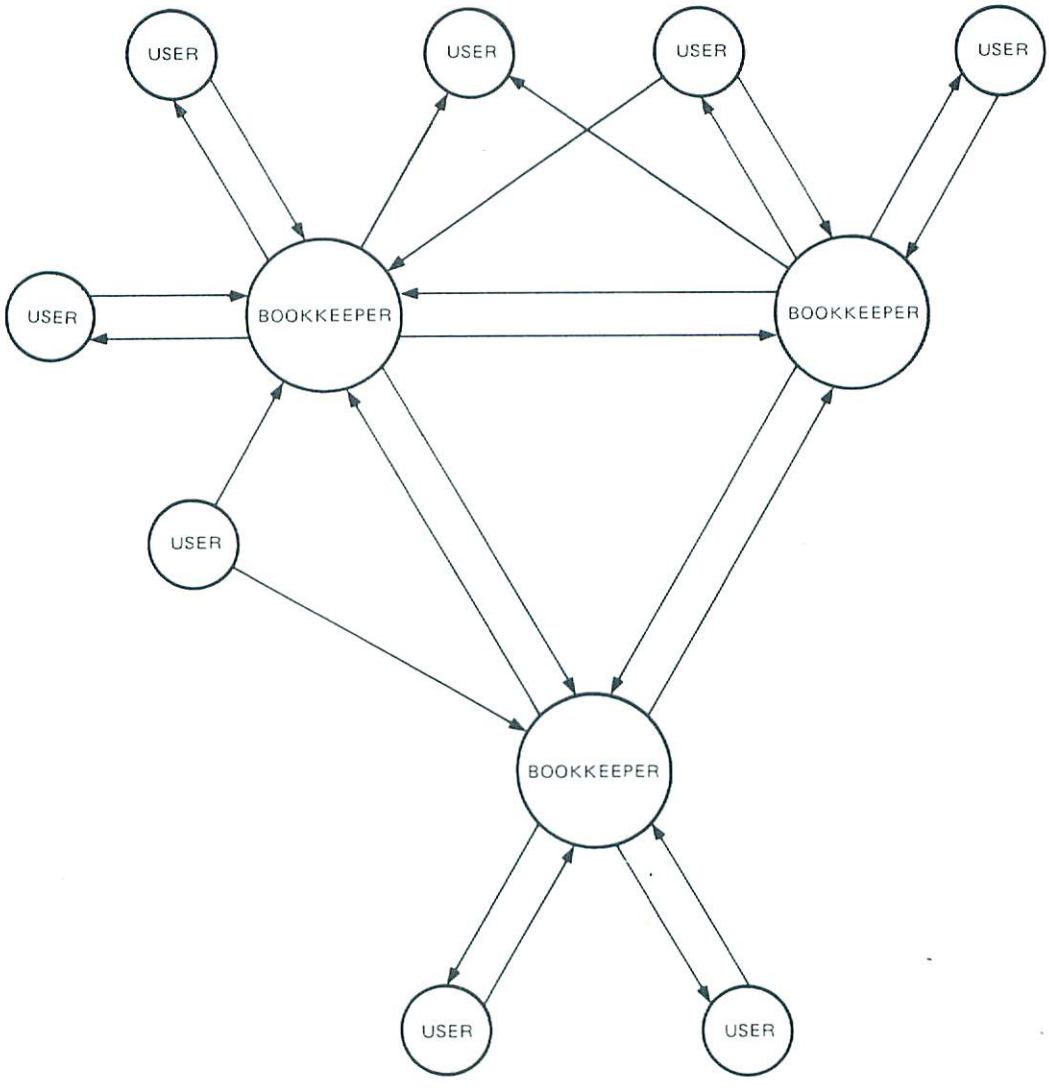

Fig. 3. A sample physical system.

processes is functioning properly. Instead of considering the entire network, it is convenient to restrict our attention to the network of bookkeepers. This will result in a correctness condition which is not completely general, and which can be satisfied only under special assumptions. What these assumptions are, and how the condition can be generalized, are discussed in Section 3.2.3.

Proper functioning of communication lines between users and bookkeepers is needed to insure that commands reach a bookkeeper soon enough – i.e., within δ seconds of when they are issued, for some δ. Since we are ignoring the problem of distributing responses to the users, we can reformulate Goals 1 and 2 as follows.

If a large enough subnetwork m of the network of bookkeepers functions properly, then any command issued at a time T which is received by any bookkeeper in m before time $T + \delta$ will be accepted, and will be executed by every bookkeeper in m before time $T + \Delta$.

Note that this condition explicitly mentions only the network of bookkeepers, and depends upon the configuration of user processes only through the parameter δ. To state the condition more precisely, we must define the exact hypothesis which the subnetwork m must satisfy.

We begin with an intuitive discussion. At any time, let an *amoeba* be a maximal strongly connected sub-

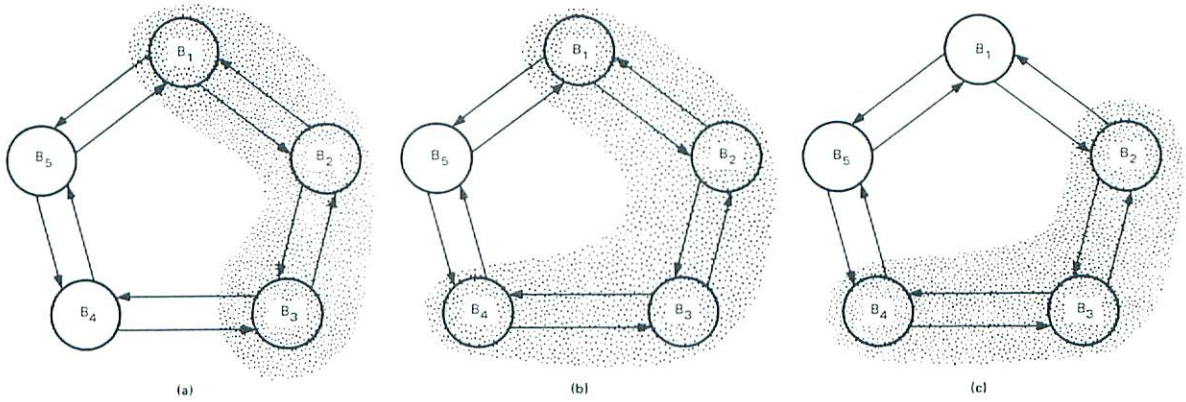

Fig. 4. The movement of the amoeba.

network of the bookkeeper network which is functioning properly. ("Functioning properly" will be defined in Section 1.5.) In fig. 4a, suppose that bookkeeper B_5 has failed and that the communication lines joining bookkeepers B_3 and B_4 have also failed. Then the shaded part of the network is an amoeba. As components fail and are repaired, an amoeba will retract and extend "pseudopods". If the communication lines between B_3 and B_4 are repaired, then the amoeba will extend a pseudopod and appear as the shaded part of fig. 4b. If B_1 now fails, then the amoeba will retract a pseudopod and appear as in fig. 4c. The subnetwork m in the above condition is an amoeba.

If the bookkeepers in an amoeba are to continue executing the user machine, then the amoeba must be large enough. In the network of fig. 4, an amoeba containing only two bookkeepers cannot be allowed to execute commands. Otherwise, there could be two separate amoebas independently executing commands, and they could issue conflicting responses. This cannot happen if we require the amoeba m to contain three bookkeepers.

The above condition only considers a static amoeba m, and does not consider the movement of the amoeba. We require the additional hypothesis that the amoeba not move around too quickly. If the amoeba moves as in fig. 4, then we require that it remain in the configuration of fig. 4b for some minimum length of time Ω. It may not jump too quickly from fig. 4a to fig. 4c. There must be enough time for information to flow out of bookkeeper B_1 before bookkeepers B_2, B_3 and B_4 can operate the system by themselves. Such a requirement seems necessary, but space limitations do not permit a discussion of why this is so.

Having seen that the amoeba should be large enough and not move too quickly, we must now state these requirements more precisely. We assume that we have specified a set of *majority graphs*, where a majority graph is a strongly connected subnetwork of the bookkeeper network. We require only that any two majority graphs have at least one bookkeeper in common. The amoeba is "large enough" if it contains a majority graph.

For the network of fig. 4, we take any three contiguous bookkeepers and the communication arcs joining them to be a majority graph. For a similar ring of four bookkeepers, we might choose three majority graphs as follows: (1) B_1, B_2, (2) B_2, B_3, B_4 and (3) B_3, B_4, B_1. In any actual implementation, the choice of the majority graphs would depend upon the topology of the network, the reliability of the individual components, and the transmission delays along the various arcs.

In Section 1.5, we will define precisely what it means for (all the components of) a majority graph to be functioning properly during the time interval $[T, T']$ – the interval from time T to time T'. The hypothesis that the amoeba be large enough and not move too quickly can then be restated as follows: at any time there is a properly functioning majority graph m, and some other majority graph m' must begin functioning properly at least Ω seconds before m stops functioning properly. Letting T_0 denote the time at which the system is started initially, Goals 1 and 2 can be restated precisely as the following correctness condition.

RC. For any time T: assume that there exist times $T_1, \ldots, T_n$ with $T_0 < T_1 < \ldots < T_n < T - \Omega$ such that there is a majority graph m_i which functions properly

during $[T_i, T_{i+1} + \Omega]$ $(0 \leqslant i < n)$ *and a majority graph* m_n *which functions properly during* $[T_n, T + \Delta]$*; and let C be a command issued at time T.*

(a) If C is received by some bookkeeper in m_n *before* $T + \delta$*, then it will be accepted.*

(b) If C is accepted then it will be executed by every bookkeepers in m_n *before* $T + \Delta$.

The difference between Goals 1 and 2 lies in the type of behavior we assume for the part of the network which is not functioning properly. Note that RC implies that if two majority graphs m_n and m'_n are functioning properly during $[T_n, T + \Delta]$, then the command C will be executed by the bookkeepers in both m_n and m'_n before $T + \Delta$ if it is received by a bookkeeper in either one before $T + \delta$. Hence, RC is a precise statement of the preceding intuitive condition if the amoeba is defined to be the union of all properly functioning majority graphs.

If the hypothesis of RC becomes false for some time T, then it will remain false for all later times. In other words, RC says nothing about recovery if the failure of too many components causes the entire system to fail. In this case, we would like the system to resume operating after enough components are once again functioning properly. This problem is discussed in Section 4.4.

1.5. Proper functioning

As we have already indicated, proper functioning means doing the correct thing within a specified length of time. A transmission arc is functioning properly if messages are transmitted correctly over it with a small enough delay. A bookkeeper is functioning properly if it responds correctly and quickly enough to an event. In our algorithm, the only types of event to which a bookkeeper must respond are the receipt of a message and its own clock reaching a certain time. It will respond only by sending a message. It is convenient to include the response time as part of the transmission delay of that message, and to pretend that the bookkeeper responds instantly. We then get the following condition for a majority graph m to be functioning properly during the time interval $[T_1, T_2]$.

PF1. For every time T in $[T_1, T_2]$*:*

(a) For every bookkeeper B in m: B responds correctly and instantaneously to any event occurring in it at time T.

(b) For every arc α *in m: if* $T + \delta_\alpha < T_2$*, then a message sent over* α *at time T is received correctly before time* $T + \delta_\alpha$.

We emphasize that the instantaneous response of condition (a) is purely a matter of convention, and does not represent any physical assumption.

PF1 is a definition, not an assumption. We do not assume that a message sent over α will be received within δ_α seconds of when it is sent. If it is not, then the arc is defined not to be functioning properly. The δ_α are fixed parameters. Since response times and transmission delays will vary statistically with time, the choice of the δ_α determines how likely it is for the majority graph to be functioning properly. Increasing the δ_α increases the reliability of the majority graph. For example, we can make δ_α large enough to allow time for several retransmissions of a garbled message, thereby increasing the probability that PF1(b) holds. However, the values of δ_α appear in our algorithm, and increasing them will also increase the system response time Δ. The choice of the δ_α therefore involves a tradeoff between reliability and response time.

To simplify the exposition, we will describe our algorithm in terms of *knowledge* rather than bookkeeper states. We say that a bookkeeper "knows" some fact when it has received enough information to enable it to deduce that fact. We will not bother to describe the actual algorithm for determining the bookkeeper's state of knowledge. We assume an operation whereby a bookkeeper *broadcasts* information to all other bookkeepers. Since a bookkeeper need not receive any information it already knows, the actual messages sent in broadcasting information will depend upon each bookkeeper's knowledge of what the other bookkeepers know. In order to avoid the details of how information is broadcast, we supplement condition PF1 with the following condition.

PF2. For any time T during $[T_1, T_2 - \epsilon]$ *and any bookkeepers* B_1 *and* B_2 *in m: any information being broadcast which either originates at or reaches* B_1 *at time T will reach* B_2 *before time* $T + \epsilon$.

We have introduced a new parameter ϵ. For an appropriate implementation of the broadcasting mechanism in terms of message transmission, ϵ will be a function of the δ_α, and condition PF2 will follow from PF1. The value of ϵ does not appear in our algorithm, so it is just a descriptive parameter which allows us to find worst-case bounds for Δ and Ω. Note that PF2 introduces requirements on how infor-

mation must be routed in any implementation of the broadcast mechanism.

We define proper functioning of the majority graph m during $[T_1, T_2]$ to mean that PF1 and PF2 hold. This definition completes our specification of condition RC. Our task is to find an algorithm satisfying RC for some values of Δ and Ω. These values will be functions of ϵ and the δ_α.

We must point out that there are more implementation details hidden in PF2 than meet the eye. An example of the information that will be broadcast is the sequence C^T of all accepted user machine commands issued on or before time T. At first glance, this appears to be an impractically large amount of information to transmit. However, $C^{T'}$ will already have been broadcast for some earlier time T'. Therefore, the only new information to be transmitted is the sequence of accepted commands issued between T' and T, which will usually consist of at most one command. Hence, at second glance, there seems to be no problem.

Unfortunately, there is still a problem when restarting a failed bookkeeper, or updating one that has lost communication with the amoeba. Broadcasting C^T to a recently revived bookkeeper requires sending it all accepted commands issued from the time it failed until T. This means that if the bookkeeper is in m, then PF2 will not be satisfied for a reasonable value of ϵ until that bookkeeper has received all the information which was broadcast while it was not functioning. Had we not introduced PF2, the time needed to update a failed bookkeeper would have to be included in Ω. We will make some general remarks in Section 4.4 about restarting a failed bookkeeper, but space limitations preclude any detailed discussion of the problem.

2. A special case

2.1. Assumptions

In Section 2, we describe an algorithm for the simplest non-trivial case: the complete network of three bookkeepers shown in fig. 3. Each pair of bookkeepers B_1, B_2 and the arcs joining them form a majority graph which we denote by $\langle B_1, B_2 \rangle$. For convenience, we assume that all the δ_α are the same, and are equal to δ.

Our objective is to achieve Goal 1, so we assume that no malfunction occurs. A failed bookkeeper need never respond to an event, and a failed transmission line may lose a message or delay it arbitrarily long. However, we assume that if a bookkeeper does respond then it responds correctly, and if a message is transmitted then it is received correctly. Malfunctioning will be discussed in Section 4.3.

Our algorithm requires that each bookkeeper and user process have a clock which keeps physical time. For our special case, we assume that these are ideal clocks which are perfectly synchronized with one another, and which keep exact time. Hence, at any time T, each clock will have the value T. The case of real clocks will be considered in Section 3.2.

2.2. The synchronizer

Our assumption of perfect clocks makes it easy to implement the synchronizer of fig. 2. When a user process issues a command, it attaches to it a *timestamp* equal to the current time (which it reads from its clock). We let $T{:}C$ denote the command C with the timestamp T. The user issues the command by sending the message "$T{:}C$" to one or more bookkeepers.

The synchronizer is implemented by letting the command $T{:}C$ precede the command $T'{:}C'$ in the user machine's command sequence if $T < T'$. If $T = T'$, then we assume some method for defining which of the two commands comes first. The method can be arbitrary, but all bookkeepers must independently make the same choice. We then have a total ordering of all commands, which is precisely what the synchronizer must produce. Condition SC is satisfied because we are assuming perfect clocks.

We let C^T denote the set of all *accepted* user machine commands $T'{:}C'$ with $T' < T$. In order to execute a user machine command, it is sufficient (but not always necessary) for a bookkeeper to know the complete sequence of user commands up to and including that command. Hence, a bookkeeper can execute an accepted command $T{:}C$ when it knows C^T. Our task is to insure that all the bookkeepers in the amoeba know C^T before time $T + \Delta$. We assume that the time T_0 is before any commands are issued, so C^{T_0} is empty.

2.3. The algorithm

The basic idea of our algorithm is to have each bookkeeper vote on whether to accept or reject a command. A command is accepted if and only if it

receives two acceptance votes. To satisfy RC, we must design a voting algorithm so that if $\langle B_1, B_2 \rangle$ is functioning properly, then B_1 and B_2 will be able to decide by themselves whether a command is accepted or rejected. This means avoiding the situation in which one of them has voted to accept a command, the other has voted to reject it, and they do not know how the third bookkeeper voted. Such a deadlock is avoided by the following rule for casting acceptance votes.

AR1.

(a) If a bookkeeper receives a "T:C" message from a user process before time $T + \delta$, then it:

(i) votes to accept the command T:C, and

(ii) sends an "I vote to accept T:C" *message to each other bookkeeper.*

(b) If a bookkeeper receives an "I vote to accept T:C" *message before time $T + 2\delta$, then it:*

(i) votes to accept the command T:C, and

(ii) broadcasts T:C and the fact that it was accepted.

Rule AR1 states when a bookkeeper will vote to accept a command. It therefore implies which commands the bookkeeper will vote to reject. It is necessary for the bookkeeper explicitly to cast these rejection votes. At time $T + 2\delta$, the bookkeeper will no longer vote to accept any command timestamped on or before T. It can thus send messages to the other bookkeepers at time $T + 2\delta$ stating that it is now voting to reject any command timestamped on or before T which it has not already voted to accept. (In this way, it can vote to reject commands it does not even know about.) To describe this in terms of broadcasting information, we let V_B^T denote the set of all commands timestamped on or before T which bookkeeper B has voted to accept. We then make the following rule for casting rejection votes. [Part (b) of the rule will be given later.]

AR2. (a) When a bookkeeper B knows both (i) that a command T:C has been issued, and (ii) V_B^T, then it broadcasts V_B^T.

Of course, bookkeeper B knows V_B^T at time $T + 2\delta$. If a command has already been accepted, then it no longer matters whether or not B voted to accept it. Hence, if B knows that the command $T':C'$ has been accepted, then it may omit it from the set V_B^T which it broadcasts. (The fact that $T':C'$ was accepted has already been broadcast according to rule AR1(b).)

Let $C^{[T_1,T_2]}$ denote $C^{T_2} - C^{T_1}$, the set of all accepted commands $T:C$ with $T_1 < T \leqslant T_2$. Similarly, we define $V_B^{[T_1,T_2]}$ to equal $V_B^{T_2} - V_B^{T_1}$. We first prove the following result.

Proposition 1. *Assume that each bookkeeper obeys rules AR1 and AR2(a), and that the majority graph $\mathfrak{m}$ functions properly during $[T_1, T + 2\delta + 2\epsilon]$, where $T_1 < T$.*

(a) If any bookkeeper in $\mathfrak{m}$ votes to accept the command T:C, then it will be accepted.

(b) If the command T:C is accepted, then every bookkeeper in $\mathfrak{m}$ will know $C^{[T_1,T]}$ before time $T + 2\delta + 2\epsilon$.

Proof. Let $\mathfrak{m} = \langle B_1, B_2 \rangle$, let $T':C'$ be any command with $T_1 < T' \leqslant T$, and suppose that B_1 votes to accept $T':C'$. It can do so only by rule AR1, so we consider the following two cases.

Case (i): B_1 votes to accept $T':C'$ by AR1(a). In this case, B_1 will send an "*I vote to accept $T':C'$*" message to B_2 before $T' + \delta$. By PF1, B_2 will receive that message before $T' + 2\delta$. By AR1(b), B_2 will then also vote to accept $T':C'$, so it will be accepted. Rule AR1(b) also implies that B_2 will then broadcast the fact that $T':C'$ is accepted, so PF2 implies that B_1 and B_2 will both know that $T':C'$ is accepted before $T' + 2\delta + \epsilon$. Since B_1 and B_2 both learn about the command $T':C'$ before $T' + 2\delta$, AR2(a) and PF2 imply that they will both know $V_{B_1}^{T'}$ and $V_{B_2}^{T'}$ before $T' + 2\delta + \epsilon$.

Case (ii). B_1 votes to accept $T':C'$ by AR1(b). This implies that B_1 broadcasts the fact that $T':C'$ is accepted before $T' + 2\delta$, so PF2 implies that B_2 learns this fact before $T' + 2\delta + \epsilon$. By rule AR2(a), B_2 must therefore broadcast $V_{B_2}^{T'}$ before $T' + 2\delta + \epsilon$; so by PF2, B_1 learns $V_{B_2}^{T'}$ before $T' + 2\delta + 2\epsilon$. Moreover, by AR2(a), B_1 broadcasts $V_{B_2}^{T'}$ at time $T' + 2\delta$, so B_2 knows $V_{B_1}^{T'}$ before $T' + 2\delta + \epsilon$.

In both cases, $T':C'$ is accepted. Letting $T':C' = T:C$, this proves part (a). Moreover, we have also proved that for any command $T':C'$ in $V_{B_1}^{[T_1,T]} \cup V_{B_2}^{[T_1,T]}$: (1) B_1 and B_2 will know that $T':C'$ has been accepted before $T' + 2\delta + \epsilon$, and (2) B_1 and B_2 will both know $V_{B_1}^{[T_1,T']} \cup V_{B_2}^{[T_1,T']}$ before $T' + 2\delta + 2\epsilon$. But since a command cannot be accepted without an acceptance vote from either B_1 or B_2, each bookkeeper knows that $C^{[T_1,T]} \subset V_{B_1}^{[T_1,T]} \cup V_{B_2}^{[T_1,T]}$, so this proves part (b). □

Proposition 1 essentially states that the amoeba is large enough if it contains a majority graph. However, it does not permit the amoeba to move around. The problem is that rules AR1 and AR2(a) only generate messages when a command is issued. If a bookkeeper does not hear from the other bookkeepers, it could be either because of a failure or because no commands were issued. We must introduce new rules to remedy this.

First, we augment rule AR2(a) so a bookkeeper casts its rejection votes at least once every τ seconds, where $\tau \geqslant 2\delta + \epsilon$ is a new system parameter.[2] Increasing τ will decrease the number of messages that must be sent when there is no activity, but it will also increase the value of Ω in condition RC.

AR2. (b) For every time T: a bookkeeper B must broadcast $V_B^{T'}$ before time $T + \tau$ for some $T' \geqslant T$.

Rule AR2(b) allows us to prove the following result.

Proposition 2. *Assume that each bookkeeper obeys rules AR1 and AR2, and that a majority graph m functions properly during $[T_1, T + \tau + \epsilon]$, where $T_1 < T$. Then every bookkeeper in m knows $C^{[T_1, T]}$ before $T + \tau + \epsilon$.*

Proof. Let $m = \langle B_1, B_2 \rangle$, and let T' be the earliest time such that $T_1 \leqslant T' \leqslant T$ and $V_{B_1}^{[T', T]} \cup V_{B_2}^{[T', T]}$ is empty. If $T' > T_1$, then Proposition 1 implies that B_1 and B_2 both know $C^{[T_1, T']}$ before $T' + 2\delta + 2\epsilon \leqslant T + \tau + \epsilon$. If $T' = T_1$, then they trivally know $C^{[T_1, T']}$, since it is empty by definition. They will also know that $C^{[T', T]}$ is empty, and thus know $C^{[T_1, T]}$, when they know $V_{B_1}^T \cup V_{B_2}^T$. But AR2(b) and the hypothesis that PF2 holds implies that this will occur before $T + \tau + \epsilon$. □

Finally, we add the following rule to assure that knowledge of C^T is broadcast.

AR3. For every time T: a bookkeeper broadcasts C^T as soon as it knows it.

We will not bother to write the precise rule for when a bookkeeper knows C^T. We merely observe that a bookkeeper knows C^T if it knows C^{T_1} and $C^{[T_1, T]}$. We can now prove the correctness of our algorithm. (Remember that we are assuming that there may only be failures, but not malfunctions.)

Theorem. *A network of three bookkeepers obeying rules AR1–3 satisfies condition RC with $\Delta = 2\delta + 2\epsilon$, and $\Omega = \tau + 2\epsilon$.*

Proof. Part (a) of RC follows immediately from rule AR1(a) and part (a) of Proposition 1. We now prove part (b). We can obviously assume that $m_i \neq m_{i+1}$. We first show that for each $j = 0, \ldots, n$: every bookkeeper in m_j knows C^{T_j} before $T_j + \tau + 2\epsilon$. The proof is by induction on j. Since C^{T_0} is empty by definition, the result is trivial for $j = 0$. Assume that it is true for all $i < j$, with $1 \leqslant j \leqslant n$. Let $m_{j-1} = \langle B_1, B_2 \rangle$ and $m_j = \langle B_2, B_3 \rangle$. By the induction hypothesis, B_2 knows $C^{T_{j-1}}$ before $T_{j-1} + \tau + 2\epsilon$. By Proposition 2, B_2 knows $C^{[T_{j-1}, T_j]}$ before $T_j + \tau + \epsilon$. This implies that B_2 knows C^{T_j} before *maximum* $(T_{j-1} + \tau + 2\epsilon, T_j + \tau + \epsilon) \leqslant T_j + \tau + 2\epsilon$. Thus, we need only show that B_3 knows C^{T_j} before $T_j + \tau + 2\epsilon$. We consider two cases.

Case (i): $j = 1$ or $T_{j-1} \leqslant T_j - \epsilon$. In this case, B_2 knows both $C^{T_{j-1}}$ and $C^{[T_{j-1}, T_j]}$, and thus knows C^{T_j}, before $T_j + \tau + \epsilon$. Rule 3 thus implies that B_2 broadcasts C^{T_j} before $T_j + \tau + \epsilon$, so the hypothesis of RC and PF2 imply that B_3 learns C^{T_j} before $T_j + \tau + 2\epsilon$.

Case (ii): $j > 1$ and $T_{j-1} > T_j - \epsilon$. In this case, the hypothesis of RC implies that m_{j-2} functions properly during $[T_{j-2}, T_j + \tau + \epsilon]$. By the induction hypothesis, each bookkeeper in m_{j-2} knows $C^{T_{j-2}}$ before $T_{j-2} + \tau + 2\epsilon$, and hence by Proposition 2 knows C^{T_j} before *maximum* $(T_{j-2} + \tau + 2\epsilon, T_j + \tau + \epsilon) \leqslant T_j + \tau + 2\epsilon$. But $m_{j-2} \neq m_{j-1} \neq m_j$ implies that B_3 is in m_{j-2}, completing the induction proof.

Letting $j = n$, we have thus shown that each bookkeeper in m_n knows C^{T_n} before time $T_n + \Omega < T$. Proposition 1 implies that each bookkeeper in m_n knows $C^{[T_n, T]}$ before $T + \Delta$. Hence, each bookkeeper in m_n knows C^T, and will therefore execute $T{:}C$, before $T + \Delta$. □

2.4. Discussion of the algorithm

It is an interesting feature of our algorithm that a bookkeeper never asks whether a transmission line or another bookkeeper is functioning properly – or even whether it is itself functioning properly. It just blindly follows the rules, confident that condition RC

[2] The dependence of τ on ϵ seems to contradict our earlier statement that the algorithm is independent of the value of ϵ. However, the constraint that $\tau \geqslant 2\delta + \epsilon$ is introduced only to simplify our expressions of Δ and Ω as functions of ϵ. This constraint should hold in any practical implementation.

will be satisfied. It is easy to detect "dead" components which do nothing for a long period of time, and some mechanism for detecting them should be included so they can be repaired. However, this is an implementation detail which does not concern us. The important fact is that transient failures, such as lost messages, do not have to be detected. (Of course, in the implementation of the broadcasting mechanism, a process must keep trying to send information until it learns that it was received.)

Stating our rules in terms of broadcasting information simplified the analysis of the algorithm. However, to understand the actual behavior of the algorithm, we must consider how it is implemented in terms of message sending. To do this without introducing a mass of uninteresting details, we make two simplifying assumptions: (1) every message is eventually received, and (2) messages sent over any single arc are received in the same order as they are sent. The first assumption eliminates the need for special procedures to restart a failed bookkeeper, and the second assumption avoids having to keep track of message numbers.

Implementation of rules AR1–3 under these assumptions is straightforward, and the details will be left to the reader. We merely mention that such an implementation has the following properties.

(i) Broadcasting V_B^T is done by having B send any message to the other bookkeepers timestamped on or later than $T + 2\delta$.

(ii) Broadcasting C^T involves sending the message "*I have already notified you of all accepted commands timestamped on or before T.*"

Suppose that a command is sent to only one bookkeeper. A naive count indicates that our algorithm generates 26 messages between the bookkeepers in order to execute the command. By eliminating messages containing unnecessary information, and combining the messages generated by rules AR1(b) and AR2(a), one finds that at most 14 messages need to be sent. This is still a large number. However, only four of these messages contain the command T:C; the rest contain only the timestamp T plus a few bits of information. Hence, the total amount of information being sent is not very large. The total number of messages can be reduced by combining several messages (generated by different commands or by other activity) into a single message. This allows one to decrease the total number of messages at the cost of increasing ϵ (because information is buffered instead of being sent immediately), thereby increasing the system's response time Δ. A more thorough discussion of such implementation details is beyond the scope of this paper.

3. The general algorithm

3.1. An arbitrary network

We have considered our algorithm in detail for the special case of a three bookkeeper network. We now describe how it can be generalized to a collection of majority graphs in an arbitrary network. We use the same basic idea of having each bookkeeper vote on whether or not to accept a command. The obvious generalization of requiring two acceptance votes is to let a command be accepted if and only if every bookkeeper in some majority graph votes to accept it. Voting rule AR1 must therefore be generalized, but we have stated rules AR2 and AR3 in a sufficiently general form so they do not have to be changed.

Rule AR1 was designed so that if one bookkeeper in a properly functioning majority graph votes to accept a command, then that command will be accepted. We must generalize AR1 so that this property holds for an arbitrary network. This is done as follows. If a bookkeeper B_1 receives a command T:C from a user before time $T + \delta$, then it initiates a cascade of voting messages. Each message in the cascade is of the form "B_1 *and* B_2 *and ... and* B_k *have voted to accept* T:C", which is sent by bookkeeper B_k. If bookkeeper B_{k+1} receives this message early enough, then it votes to accept T:C. If the set of bookkeepers $B_1, ..., B_{k+1}$ does not contain all the bookkeepers in some majority graph, then B_{k+1} continues the cascade by sending the message "B_1 *and ... and* B_{k+1} *have voted to accepted* T:C" to one or more other bookkeepers. The rules for generating this cascade must be such that for any majority graph m, proper functioning of m will guarantee acceptance of the command if any single bookkeeper in m votes to accept it.

As an example, we consider the five bookkeeper ring of fig. 4 in which any three adjacent bookkeepers and the arcs joining them form a majority graph. Fig. 5 defines the message cascade generated by a command T:C arriving at bookkeeper B_1, where v denotes *has (have) voted to accept.* Each solid circle represents the casting of an acceptance vote. Let δ_α be the parameter defined by PF1 for the arc α from B_1 to B_2. Bookkeeper B_2 casts its acceptance vote

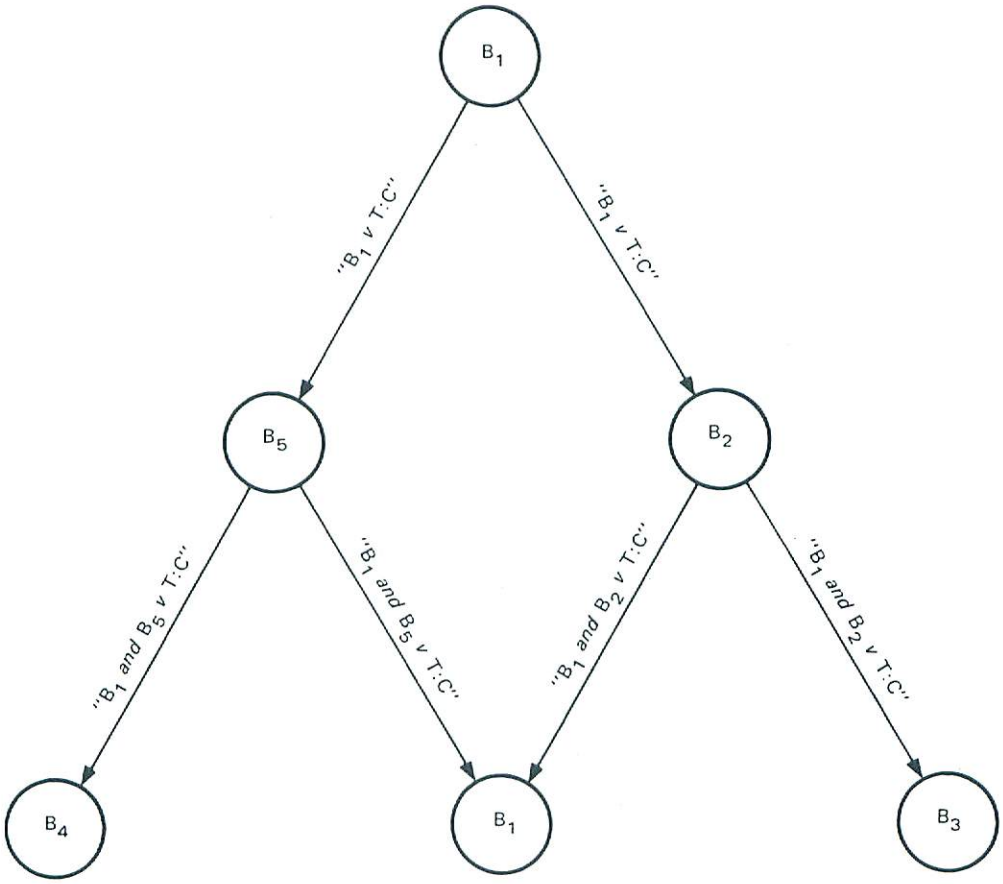

Fig. 5. A voting message cascade.

and sends its "B_1 *and* B_2 υ T:C" messages if it receives the message "B_1 υ T:C" before time $T + \delta + \delta_\alpha$. Similarly, B_3 votes to accept T:C if it receives the message "B_1 *and* B_2 υ T:C" before $T + \delta + \delta_\alpha + \delta_\beta$, where β is the arc from B_2 to B_3.

Observe that if B_1 votes to accept the command, then the proper functioning of any of the three majority graphs containing B_1 guarantees that it will be accepted. Moreover, suppose that the majority graph composed of B_2, B_3, and B_4 is functioning properly, and the "B_1 υ T:C" message reaches B_2 before $T + \delta + \delta_\alpha$. Then the command will be accepted regardless of whether B_1 is functioning properly. (The proper generation and transmission of the "B_1 υ T:C" message could have been a fluke.)

There will be a similar message cascade for each of the other four bookkeepers. If the same command may be sent to more than one bookkeeper, then how a bookkeeper will respond to a message from one cascade may be changed by the receipt of a message from another cascade initiated by the same command. The idea is simple but the details are complicated.

The generalization of rule AR1(b) is simple. As soon as a bookkeeper learns that the command T:C has been accepted, it broadcasts T:C and the fact that it was accepted to all other bookkeepers. In the cascade of fig. 5, B_3 or B_4 will know that T:C has been accepted when it casts its acceptance vote. Bookkeeper B_1 will know that T:C has been accepted if it receives the two incoming messages to the dashed node of fig. 5. Note that these two messages are needed to insure that some bookkeeper learns of T:C's acceptance in case B_3 and B_4 have failed.

One can define a message cascade for an arbitrary network, and state the conditions it must satisfy in terms of the set of majority graphs. One can then give a precise generalization of rule AR1, and prove the correctness of the resulting algorithm by proving the appropriate generalizations of the propositions and

theorem of Section 2.3. However, this requires developing quite a bit of formalism, and is rather tedious. If the reader understands the basic idea, then for any given network and collection of majority graphs he should be able to design a suitable message cascade and prove that the resulting algorithm satisfies condition RC.

3.2. Real clocks

So far, our algorithm has been described in terms of perfect clocks, all ticking in unison at precisely the correct rate. However, real clocks are not perfect, and they run at only approximately the same rate. To keep the times indicated by different clocks from drifting arbitrarily far apart, there must be some method for synchronizing them. Moreover, real clocks can malfunction. [3] We want the system to function correctly despite the malfunctioning of individual clocks. We discuss the problem of using real clocks in stages: first discussing how clocks are synchronized, then considering successively less perfect clocks.

3.2.1. Synchronizing the clocks

Let $C_i(T)$ denote the time indicated by process i's clock at real time T. For convenience, we assume that the clock is continuously advanced by some mechanism at the rate $dC_i(T)/dT$. (A discrete clock can be thought of as a continuous one in which there is an error of up to $\frac{1}{2}$ tick when reading it.) For a perfect clock, $dC_i(T)/dT$ always equals 1.

The method for synchronizing clocks was discussed at length in [1]. Whenever a process sends a message, it affixes to it a timestamp containing the current value of its clock. If a process receives a message with a timestamp later than the current value of its clock, then it advances its clock (discontinuously) to read later than that timestamp. We say that the clock C_i is functioning properly if $|dC_i(T)/dT - 1|$ is less than some fixed parameter κ, and the algorithm for discontinuously advancing C_i is executed correctly.

It is shown in [1] that if clocks are synchronized in this way, then our method of ordering commands by their timestamps satisfies condition SC of Section 1.2.2. We also derived in [1] the following bound on how far apart the readings of different clocks could become. Assume that:

(i) Every clock functions properly.

(ii) For every arc, at least once every σ seconds some message is transmitted over that arc with an uncertainty in its transmission delay of at most ξ seconds.

(iii) For every pair of processes, there is a path from one to the other containing at most D arcs.

Then the following approximate inequality holds for all i, j and T.

$$|C_i(T) - C_j(T)| \lesssim D(2\kappa\sigma + \xi)\,. \tag{3-1}$$

3.2.2. Almost perfect clocks

We now assume that clocks never malfunction, so we can use the above synchronization method and obtain the inequality (3-1) for some very small value of κ. (In practice, one can easily obtain clocks for which $\kappa < 10^{-6}$.) In other words, we assume that even if a process fails, its clock keeps running. The parameter σ represents the maximum "down time" of any process or transmission line. We assume that $\kappa\sigma$ is small – at most of the same order of magnitude as ξ. Then (3-1) gives a reasonably small upper bound on the amount by which any two processes' clocks can differ. It is then easy to modify our algorithm as follows to compensate for this difference.

For the three bookkeeper case, we use (3-1) to find quantities δ' and δ'' with $\delta < \delta' < \delta''$ such that for all processes i and j (including user processes) and all T, the following relations hold.

$$C_i(T + \delta) \leqslant C_j(T) + \delta'\,,$$
$$C_i(T) + 2\delta' \leqslant C_j(T + 2\delta'')\,. \tag{3-2}$$

If we restate our rules in the obvious way in terms of clock times, replace δ by δ' in rule AR1, and replace τ by $(1 - \kappa)\tau$ in rule AR2(b), then it can be shown that the propositions and theorem of Section 2.3 hold with δ replaced by δ''. Note that conditions PF1, PF2 and RC are stated entirely in terms of actual times, not clock times. A correctness condition stated in terms of clock times can be satisfied by simply stopping all the clocks.

The algorithm for an arbitrary network can be similarly modified to work for clocks satisfying our assumption. It is a straightforward generalization of

[3] Strictly speaking, a malfunctioning clock is one which ticks at the wrong time, thereby ticking at an incorrect rate; while a failed clock is one which does not tick at all, so its rate of ticking is zero. We can regard a failed clock as one which is malfunctioning in a special way.

the method used for three bookkeepers, and will not be discussed.

3.2.3. Monotonic clocks

We now consider clocks which can malfunction by running at arbitrary rates. However, we assume that they always increase monotonically and never run backwards. We want the system to function correctly so long as a large enough portion of the network of processes function properly, where proper functioning includes the proper functioning of its processes' clocks. Since the proper functioning of input processes' clocks is important, we cannot prove a correctness condition such as RC which only assumes the proper functioning of bookkeepers. Condition RC must be modified to require that the command C be issued by a user process in some properly functioning network containing the majority graph m_n. The idea is straightforward, and we will omit the details.

Simply modifying RC does not solve our problem. The difficulty caused by malfunctioning clocks is illustrated by the following example. Suppose that in the three bookkeeper case all the bookkeepers are initially functioning properly, and the following scenario occurs.

1. B_1 receives a command timestamped T:C which it votes to accept, and sends an "*I vote to accept* T:C" message to B_2 and B_3.
2. B_3's clock malfunctions by jumping ahead, and it sends some message timestamped $T' >> T$ to B_2.
3. B_2 receives B_3's message (before receiving B_1's) and advances its clock to T', thereby implicity voting to reject T:C.
4. B_3 fails (perhaps shut down by a malfunction detector).

We then have the situation in which B_1 has voted to accept T:C, B_2 has voted to reject it, and B_3 has failed before informing the other bookkeepers of its vote. Hence, B_1 and B_2 will not know if T:C is accepted until B_3 is repaired.

A rather complicated modification to our algorithm is needed to enable it to cope with malfunctioning clocks. It requires each bookkeeper to maintain several clocks. These clocks all run at the same rate, but they have different rules for synchronizing with other processes' clocks. We indicate how this can be done in the three bookkeeper case. To avoid having to modify condition RC, we make the simplifying assumption that each user process communicates with only one bookkeeper, and its clock is perfectly synchronized with that bookkeeper's clock. (For example, the user process and the bookkeeper could be located at the same site, and use the same physical clock.) Each bookkeeper B_i has one clock C_i which it uses in applying rule AR1(a), and a clock C_i^k which it uses in applying rule AR1(b) to messages from bookkeeper B_k. Instead of (3-2), we then need the following relations to hold for all k and T whenever B_i and B_j are in a properly functioning majority graph.

$$C_i^j(T+\delta) \leqslant C_j(T) + \delta' ,$$

$$C_j(T) + 2\delta' \leqslant C_i^k(T + 2\delta'') .$$

These relations can be maintained despite the malfunctioning of the third bookkeeper's clock by synchronizing all the clocks as before except for the following case. If $i \neq j \neq k$, then bookkeeper B_i does not advance C_i^j immediately after receiving a later timestamp from B_k. Instead, it sends a message to B_j timestamped according to C_i and waits for $2\delta(1+\kappa)$ seconds before advancing C_i^j.

The precise details of how this is done are beyond the scope of this paper. We intend to discuss the method in a more general context in a later paper.

3.2.4. Non-monotonic clocks

Finally, we consider the case in which a malfunctioning clock need not increase monotonically, but may jump backwards. This creates no problem in synchronizing the clocks. Only clocks which run too fast cause trouble; clocks which run slow are harmless. However, when a bookkeeper's clock jumps backwards, it means that the bookkeeper has forgotten the clock's previous higher reading. This is likely to imply that the bookkeeper has also forgotten about other actions it had taken. For example, it might have voted to reject certain commands by rule AR2. A backward running clock could cause it to vote to accept commands it had previously voted to reject. Such a lapse of memory must be treated as a bookkeeper malfunction. Malfunctions will be discussed in Section 4.3.

4. The total system

We have described an algorithm for implementing a given user machine which achieves Goal 1. We now want to consider how this algorithm can be used as part of a reliable total system. Since different systems

will vary widely in their details, our discussion will be limited to a few general observations.

4.1. The user machine clock

User machine commands are executed in order of increasing timestamp. This allows us to define an abstract *user machine clock* such that the command $T{:}C$ is executed at *user machine time* T. For convenience, we assume that the time T is an integer. We define the user machine clock to be part of the user machine state, and let $t(S)$ denote the time of state S. The execution mapping $\boldsymbol{e}$ defined in Section 1.2.1 is then restricted by the condition that $\boldsymbol{e}(T{:}C,\ S) = (R,\ S')$ only if $T = t(S) = t(S')$. I.e., a command is executed only when its timestamp equals the user machine time, and its execution does not change the user machine time.

We also define a mapping $\boldsymbol{i}$: $\boldsymbol{S} \rightarrow \boldsymbol{S} \times \boldsymbol{R}$ such that if $\boldsymbol{i}(S) = (S',\ R)$, then $t(S') = t(S) + 1$. The mapping $\boldsymbol{i}$ specifies what happens when the user machine clock ticks. If the user machine is in the state S at time $t(S)$, then $\boldsymbol{i}(S) = (S',\ R)$ means that at time $t(S) + 1$ the user machine goes into state S' and generates the response R. (R will usually be null.) We have thereby introduced the possibility of the user machine performing a spontaneous action at a certain time. For example, we can define $\boldsymbol{i}$ for an airline reservation system so that all unconfirmed reservations are automatically cancelled 24 h before the scheduled departure time.

Such spontaneous user machine actions may be necessary for the entire system to achieve Goal 1. In a distributed file system, Goal 1 may require that a failed user not tie up any files. Proper functioning of a user machine with ordinary *acquire* and *release* commands does not suffice, since a user could acquire a file and fail without releasing it. The user machine must be designed so that a file is automatically released at some fixed time after it is acquired.

Suppose we require that a bookkeeper always wait until time $T + \Delta$ (on its clock) before performing any user machine action at user machine time T. With perfect clocks, we then know that every bookkeeper in a properly functioning majority graph executes the user machine in real time so that the user machine's clock reads precisely Δ seconds less than the correct time. [4] For the case of almost perfect clocks discussed in Section 3.2.2, the different bookkeepers will execute the user machine in almost perfect synchrony – at any moment, the discrepancy in their clock readings will be less than some constant.

4.2. Queries

Each bookkeeper must maintain its own copy of the user machine state. Multiple copies of data can reduce transmission costs by having each user access the nearest copy. We have required that a user machine command be executed by every bookkeeper. This was necessary because a command might change the user machine state. However, there are certain commands, called *queries*, which cannot change the user machine state. The airline reservation system command "*How many seats are left on Flight 123 for 2/7/80?*" is an example of a query. A query need only be processed by a single bookkeeper. In general, the use of multiple bookkeepers can reduce transmission costs only if most data transmission is the result of queries.

We list three ways in which a bookkeeper can handle queries. The choice of which to use will depend upon the needs of the particular system.

1. The query can be given a timestamp T, and the response based upon the user machine state at time T. For example, the query might be "*12:01 PM 1/31/80: How many seats are left on Flight 123 for 2/7/80?*" The response would then be "*27 seats left*", meaning that there were 27 seats left at user machine time 12:01 PM 1/31/80. Note, however, that a response to the query $T{:}Q$ by time $T + \Delta$ cannot be guaranteed unless the other bookkeepers are informed that a query was issued at time T. If the query is received after the bookkeeper has forgotten what the user machine state was at time T, then it must be rejected.
2. The query can be issued without a timestamp, and the response based upon the latest user machine state which the bookkeeper knows. For example, the query "*How many seats are left . . .?*" issued at 12:01 PM on 1/31/80 might generate the response "*At 11:59 AM 1/31/80 there were 29 seats left.*" Unlike method 1, the response can be generated as soon as the query is received. The query need never be rejected.
3. The query can be issued without a timestamp, and the response based upon the latest information available to the bookkeeper. For example, suppose the bookkeeper receives the query "*How many*

[4] We must modify AR2(a) so that the bookkeeper B also broadcasts V_B^T when it knows that a spontaneous user machine action should take place at time T.

seats are left . . .?" at 12:04 PM 1/31/80 (on its clock); it knew that there were exactly 29 seats left at user machine time 11:59 AM; and it had received commands timestamped between 11:59 AM and 12:03 PM requesting two seats. It could guess that those commands would be accepted and there would probably be no further commands timestamped before 12:03 requesting seats on that flight. It would then respond as follows: "*There are probably 27 seats left as of 12:03 PM 1/31/80.*" As with method 2, the response can be issued when the query is received, and no query need be rejected.

4.3. Goal 2: malfunction and security

The use of multiple bookkeepers introduces considerable redundancy. We would like this redundancy to prevent the malfunctioning of individual bookkeepers or transmission lines from causing errors. However, without further precautions, multiple bookkeepers could mean multiple sources of error. Rather than restricting ourselves to bookkeepers errors, we will consider the larger problem of making the total system satisfy Goal 2.

The execution of commands procedes in three logical steps: (1) the user transmits a command to the user machine, (2) the user machine executes the command, and (3) the response is transmitted to the user(s). This leads to three possible types of error: (1) incorrect input command, (2) incorrect user machine execution, and (3) incorrect transmission of the response. These three types of error will be considered separately. First, however, we discuss the relation between Goal 2 and security.

4.3.1. Security

A system is secure if it is very difficult to cause it to perform an unauthorized operation or to divulge information without authorization. [5] To get the system to perform an unauthorized operation, a "knave" must cause some part of the system to perform incorrectly – in other words, he must generate a malfunction. The only difference between knavery and ordinary malfunctioning is that a knave may cause misbehavior which would be unlikely to occur because of natural malfunctioning. Hence, the only difference between preventing unauthorized operations and achieving Goal 2 is the class of errors one tries to handle. Although they may require drastically different implementations, the two problems are conceptually the same. For example, transmission errors can be detected by introducing redundancy into the messages. However, ordinary redundancy checks offer no protection against a knave trying to intercept and modify a message. To guard against knavery, the redundancy check must be a cryptographic message authentification mechanism, as discussed in [2].

Preventing the unauthorized divulging of information is a different problem. Whereas redundancy can help prevent unauthorized operations, the fact that multiple copies of the information are needed makes it harder to guard that information. Each copy must be guarded just as if it were the only one, so no new techniques are required. We will therefore not discuss this problem.

4.3.2. Incorrect input commands

An error can occur in an input command, or a spurious command can be generated, by a malfunction in either (i) a user process, (ii) a transmission line, or (iii) a bookkeeper. For example, a malfunctioning bookkeeper could generate an "*I vote to accept T:C*" message for a non-existent command *T:C*. The latter two sources of error are handled by adding redundancy to the commands, so that an erroneous command has a high probability of being recognized. The user machine is designed so that an erroneous command does nothing more than generate the appropriate response to report the error.

Note that all bookkeepers must use the same algorithm for recognizing erroneous commands. This means that to foil a knavishly malfunctioning bookkeeper, knowledge of the authentication algorithm must not allow one to forge a command easily. Such an algorithm has recently been proposed in [5].

Redundancy checking cannot prevent an incorrect command from being generated by a user process, because the correct redundancy can be added to an erroneous command. Guarding against this possibility requires separate, redundant commands issued by different users. The user machine can be designed so that certain critical operations are performed only after the appropriate commands have been issued by several different users within some fixed length of time. The introduction of incorrect commands to execute such a critical operation would then require

[5] We ignore the problem of insuring that the performance of an authorized operation cannot cause a security violation. Strictly speaking, that is a problem of program correctness and not of security.

the simultaneous malfunction of several user processes.

4.3.3. Incorrect user machine execution

Each bookkeeper separately simulates the execution of the user machine. A malfunctioning bookkeeper obviously cannot be expected to perform this simulation correctly. However, we must prevent it from causing the other bookkeepers to make errors. More precisely, we want to guarantee that every bookkeeper in a properly functioning majority graph will correctly execute user machine commands despite the malfunctioning of bookkeepers or transmission lines outside that majority graph.

To illustrate the nature of the problem, consider the following scenario for the three bookkeeper case in which bookkeepers B_1 and B_2 are functioning properly but B_3 is malfunctioning.

1. B_3 sends an "*I vote to accept* T:C" message to B_1, but not to B_2.
2. B_1 receives B_3's message, votes to accept T:C, and broadcasts the fact that T:C is accepted (by AR1(b)).
3. B_2, not having heard about T:C, implicitly votes at time $T + 2\delta$ to reject it.
4. B_3 (incorrectly executing AR2(b)) sends B_2 a message stating that it voted to reject T:C.
5. (a) B_2 receives B_1's message stating that T:C is accepted.
   (b) B_2 receives B_3's message stating that it voted to reject T:C, and concludes that T:C is rejected because it received two rejection votes.

At this point, B_2 knows that there has been a malfunction. However, if B_3 persists in telling B_1 that it voted to accept T:C and telling B_2 that it voted to reject T:C, then there is no way for B_2 to decide whether it is B_1 or B_3 that is malfunctioning. Similarly, B_1 will not know whether B_2 or B_3 is malfunctioning.

To solve this problem, we first recognize that the information transmitted between bookkeepers can all be expressed in terms of acceptance and rejection votes. In the above example, the information that T:C is accepted (broadcast by B_1) was based upon the votes by B_1 and B_3 to accept T:C. We can then introduce the following rules for the bookkeepers to follow.

MR1. All information is transmitted by sending acceptance and rejection votes, rather than being summarized in any way. For example, the message "B_1 and B_2 v T:C" in the cascade of fig. 5 includes the actual votes generated by B_1 and B_2.

MR2. Each bookkeeper redundantly encodes its votes in such a way that it is highly improbably for the malfunctioning of a transmission line or another bookkeeper to generate a correctly encoded incorrect vote. For simplicity, we consider such a highly improbable event to be impossible.

MR3. A bookkeeper ignores any information or voting message containing a vote which is incorrectly encoded.

MR4. If a bookkeeper receives validly encoded acceptance and rejection votes for the same command by the same bookkeeper, then it ignores the rejection vote.

The reader can verify that if the bookkeepers obey rules MR1–MR4, then Propositions 1 and 2 of Section 2.3 hold regardless of malfunctions outside of the properly functioning majority graph m (provided that the other assumptions of Section 2 still hold). An analogous result will also be true for the more general algorithms mentioned in Section 3. Propositions 1 and 2 imply that the bookkeepers in m know $C^{[T_1, T]}$ soon enough. The obvious next step is to insure that they also know C^{T_1}. Unfortunately, this is not possible for the following reason. By MR4, a malfunctioning bookkeeper can change a rejection vote on some old command T':C' to an acceptance vote. If $T' < T_1$ and the bookkeepers in m originally knew that T':C' was rejected, then the changed vote might leave them unable to decide whether T':C' is rejected or accepted.[6] The solution to this problem is to insure that the following condition is satisfied for some parameter τ'.

CC. If $T_2 > T_1 + \tau'$, then $C^{[T_1, T_2]}$ by itself determines the user machine state at time T_2 – regardless of what commands are in C^{T_1}.

Note that CC is a condition on the command issued by the users. We will describe in Section 4.4 how this condition can be achieved. It is easy to see that CC together with Propositions 1 and 2 of Section 2.3 imply that RC is satisfied. A careful analysis

[6] It is tempting simply to ignore such a late acceptance vote. However, we know of no reasonable way to insure that if one bookkeeper ignores it then the others will too.

reveals that if CC holds and the bookkeepers obey rules MR1–MR4, AR1 and AR2, then RC holds (for three bookkeepers with perfect clocks) with $\Delta = 2\delta + 2\epsilon$ and $\Omega = maximum\ (\tau, \tau') + 2\epsilon$ – even with malfunctioning components. A similar result holds for the more general algorithm. Thus, Goals 1 and 2 have been achieved.

4.3.4. Incorrect responses

Proper functioning of a majority graph obviously cannot prevent a malfunctioning bookkeeper or transmission line from generating incorrect responses. These incorrect responses must be detected by the user process receiving them. This requires redundancy in the implementation of the distributor – i.e., the user should receive the same response from more than one bookkeeper. The idea is simple, and we will not discuss it further.

4.4. Recovery from failure and malfunction

We have described how to implement the system so it continues to operate correctly as long as enough components function properly. No system can operate correctly if too many of its components fail or malfunction. However, the system should be able to resume correct operation after enough components have resumed functioning properly. Incorrect operation of the system defined by a user machine can be of two types:

1. *System failure*: the user machine does not execute commands.
2. *System malfunction*: a user process receives an incorrect response.

As an example of system failure in our three bookkeeper algorithm, suppose that all communication arcs between bookkeepers fail. Then B_1 could vote to accept a command $T{:}C$ while B_2 votes to reject it. We want B_1 and B_2 to be able to execute user machine commands if communication between them is restored. However, the algorithm of Section 2.3 does not insure this, since they may be unable to execute any more commands until they learn if $T{:}C$ is accepted, which requires that communication with B_3 be restored.

As an example of system malfunction, suppose that a malfunction of B_1 causes it to execute a user machine command incorrectly, and thereby get an incorrect version of the user machine state. Having an incorrect version of the user machine state can cause it to generate incorrect responses even after it has resumed functioning properly. If these incorrect responses are generated after bookkeeper B_3 fails, then there may be no way to determine that it is B_2's responses that are the correct ones.

The solution to the recovery problem lies in having a special user process called an auditor. To indicate what the auditor does, let us first suppose that the user machine state contains only a small amount of information. The auditor would issue a query to determine the user machine state at time T_1. By examining the responses from several bookkeepers, it tries to decide what the correct state was. If it cannot decide, then it asks a higher level (probably human) auditor for help. If it decides that the state was S_1, then at time T_2 it issues the following *checkpoint* command: "T_2: *the state at* T_1 *was* S_1". Executing this checkpoint command sets the user machine state to the value it would have if all the other commands in $C^{[T_1, T_2]}$ were executed starting with the user machine state equal to S_1. If the auditor issues such checkpoint commands frequently enough, then condition CC of Section 4.3.3 will be satisfied.

The purpose of the checkpoint is to insure that knowledge of $C^{[T_1, T_2]}$ allows a bookkeeper to determine the user machine state at time T_2. If the user machine state contains too much information, then this is not practical. However, the auditor can issue commands such that knowledge of $C^{[T_1, T_2]}$ allows a bookkeeper to deduce the user machine state at time T_2 with a very high probability of being correct. Let F be some hash coding function on the user machine state. The auditor can issue the following, more general type of checkpoint command: "T_2: *the state S at* T_1 *satisfied* $F(S) = \tilde{S}_1$". If a bookkeeper's version of the user machine state at time T_1 is S_1 and $F(S_1) = \tilde{S}_1$. Then the bookkeeper assumes that S_1 is the correct version. Otherwise, the auditor must locate the error in that bookkeeper's version and correct it by executing appropriate user machine commands. Space limitations preclude a discussion of the details of how this can be done.

It is clear that this type of checkpointing can also be used in restarting a failed bookkeeper. The problem of a malfunctioning auditor is handled by the use of multiple auditors as discussed in Section 4.3.2. This also protects against auditor failure. In practice, there would probably be an auditor at the same physical location as each bookkeeper, being executed by the same computer.

By insuring that condition CC is met, the auditors

can eliminate the need for rules AR2(b) and AR3 of Section 2.3. Auditor commands can be executed frequently enough that AR2(b) is automatically implied by AR2(a). The information which bookkeepers send to each other by AR3 is replaced by the information they send to the auditors.

The introduction of auditors may seem to be an *ad hoc* approach. However, it is actually a natural application of the idea of a sequential user machine. Error handling procedures should be designed along with the ordinary system operations, and not added as an afterthought. By making it part of the user machine, error handling can be designed to meet the specific needs of the individual system. In a banking system, one would want a discrepancy of several dollars between bookkeepers and a discrepancy of several million dollars to be handled in different ways. Sophisticated error handling procedures are feasible because they are written for the simple sequential user machine rather than the complex network of concurrently operating physical processes.

4.5. Efficiency

Our task has been to implement a very reliable system, and we have not worried about the efficiency of the solution. With our algorithm, executing a command takes a fairly long time and requires the transmission of many messages. In practice, the details of the specific system will permit optimizations to improve the efficiency of the implementation. However, the cost of achieving such a high degree of reliability will still be prohibitive for many systems. For example, it would probably not be feasible to implement a real airline reservation system in the way we have described. In this case, we must be content with a less reliable system. Errors and transient failures will have to be tolerated, so long as the system can recover gracefully from them.

Even though the system is not expected to be perfectly reliable, we still want to be sure that it satisfies some reliability conditions. This means that we must state precisely what reliability conditions it should satisfy, and guarantee that the implementation does satisfy these conditions. To do this, we can first define the execution of the system in terms of a special user machine called the *kernel* machine, whose only users are the auditor processes introduced in Section 4.4. We can then use our algorithms to implement a very reliable kernel machine.

To illustrate this approach, suppose that we wish to implement an airline reservation system with computers at three sites. A more efficient but less reliable system is obtained by having only one of the computer actively processing reservations, while the other two act as backup and periodically receive updating messages from the active computer. The state of the kernel machine would define which computer is the currently active one. A failure or malfunction of the active computer would be detected by the auditors, which would then issue kernel machine commands to activate one of the backup computers. The details of how this is done are non-trivial. However, the reliability of the resulting algorithm can be analyzed because it is defined in terms of a *sequential* kernel machine whose reliability properties are known. The efficiency of the kernel machine's implementation is of little concern, since kernel machine commands are executed infrequently.

In general, our algorithm will be practical only when commands other than queries are issued infrequently. If that is not true for the entire system, then we must define a system kernel for which it is true. We can then use our algorithm to implement a very reliable kernel. In the above implementation of an airline reservation system, each computer continually queries the kernel machine to find out which is the active computer; but commands to change the kernel machine state are infrequent. In the distributed file system discussed in Section 1.2.1, the kernel machine handled only the less frequent operations of acquiring and releasing files, but not the operations of reading and writing the files.

5. Conclusion

Our goal has been a method for implementing a reliable system with a network of independent processes. We have concentrated on the problem of failure, and discussed malfunctioning only briefly. The problem is difficult because no special assumptions were made about how a component can fail. Making such assumptions can allow simpler, more efficient solutions. For example, several earlier papers have assumed that communication lines never fail and that a failed process is restarted at a specified point in its algorithm [3,4]. What we have done is to show that the problem can be solved without this kind of assumption.

Experience has taught us that multiprocess algorithms are difficult to write, and tend to have subtle

time-dependent errors. No such algorithm can be believed without a rigorous proof of its correctness. We must be especially careful with *distributed* multiprocess algorithms, since we are less familiar with them. It would be foolhardy to try constructing a reliable distributed multiprocess system without basing it upon an algorithm which has been proved correct. Yet, rigorous correctness proofs for complex multiprocess algorithms seem unfeasible at present. Our solution to this problem was to define the system, including error detection and recovery mechanisms, in terms of a sequential user machine. We cannot overemphasize the importance of this approach. By reducing the design of the system to a sequential programming problem, we achieved an enormous simplification.

We have described how the user machine can be reliably implemented in the presence of component failure with a distributed multiprocess system. Many important details have been omitted for lack of space. Our algorithm can be defined precisely, and a rigorous correctness proof can be given. We have tried to indicate how this is done by considering the special case of a three bookkeeper network with perfect clocks. The algorithm for larger networks and real clocks is more complicated, but it is still simple enough to allow a rigorous correctness proof.

Our algorithm requires that all bookkeepers simulate the execution of the user machine. In most real systems, all the processes will not be performing the same function, so they will do more than just execute the identical user machine operations. However, any system will require some synchronizing kernel to enable the processes to form a single coherent system. It is this synchronizing kernel which would be defined by the user machine. The harmonious cooperation of the bookkeeper processes can then be achieved because they will execute the same user machine operations in (approximate) synchrony.

Index (reference is to section number)

References

[1] L. Lamport, Time, clocks and the ordering of events in a distributed system, Massachusetts Computer Associates, Inc., CA-7603-2911, March 29, 1976, to appear in Comm. ACM.

[2] W. Diffie and M. Hellman, Privacy and authentication: an introduction to cryptography, to appear in Proceedings of the IEEE.

[3] B. Lampson and H. Sturgis, Crash recovery in a distributed data storage system, to appear in Comm. ACM.

[4] C. Ellis, A robust algorithm for updating duplicate databases, Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977.

[5] R.L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Comm. ACM 21 (1978) 120.