Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

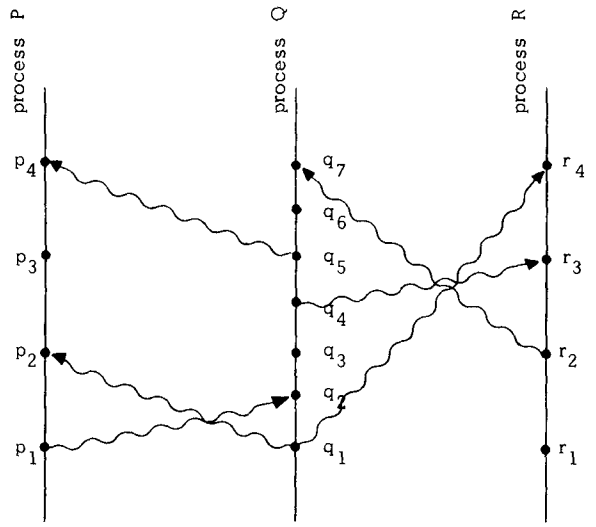
In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event a happened before an event b if a happened at an earlier time than b. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

Fig. 1.



event. We are assuming that the events of a process form a sequence, where a occurs before b in this sequence if a happens before b. In other words, a single process is defined to be a set of events with an a priori total ordering. This seems to be what is generally meant by a process.[1] It would be trivial to extend our definition to allow a process to split into distinct subprocesses, but we will not bother to do so.

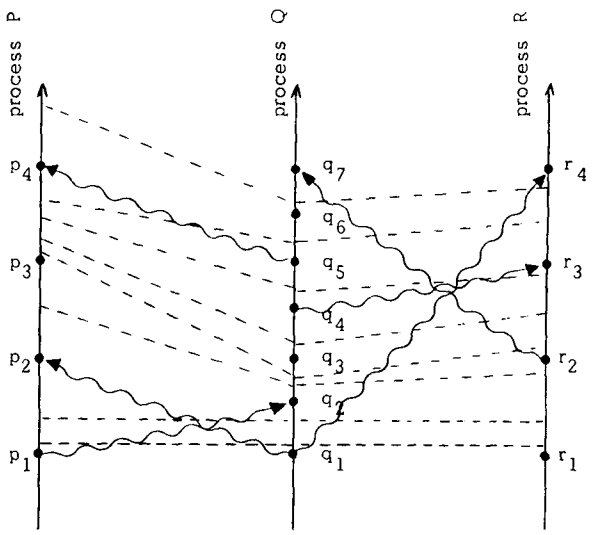
We assume that sending or receiving a message is an event in a process. We can then define the "happened before" relation, denoted by "$\rightarrow$", as follows.

Definition. The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

We assume that $a \not\rightarrow a$ for any event a. (Systems in which an event can happen before itself do not seem to be physically meaningful.) This implies that $\rightarrow$ is an irreflexive partial ordering on the set of all events in the system.

It is helpful to view this definition in terms of a "space-time diagram" such as Figure 1. The horizontal direction represents space, and the vertical direction represents time—later times being higher than earlier ones. The dots denote events, the vertical lines denote processes, and the wavy lines denote messages.[2] It is easy to see that $a \rightarrow b$ means that one can go from a to b in the diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_4$ in Figure 1.

Fig. 2.



Another way of viewing the definition is to say that $a \rightarrow b$ means that it is possible for event a to causally affect event b. Two events are concurrent if neither can causally affect the other. For example, events p_3 and q_3 of Figure 1 are concurrent. Even though we have drawn the diagram to imply that q_3 occurs at an earlier physical time than p_3, process P cannot know what process Q did at q_3 until it receives the message at p_4. (Before event p_4, P could at most know what Q was *planning* to do at q_3.)

This definition will appear quite natural to the reader familiar with the invariant space-time formulation of special relativity, as described for example in [1] or the first chapter of [2]. In relativity, the ordering of events is defined in terms of messages that *could* be sent. However, we have taken the more pragmatic approach of only considering messages that actually *are* sent. We should be able to determine if a system performed correctly by knowing only those events which *did* occur, without knowing which events *could* have occurred.

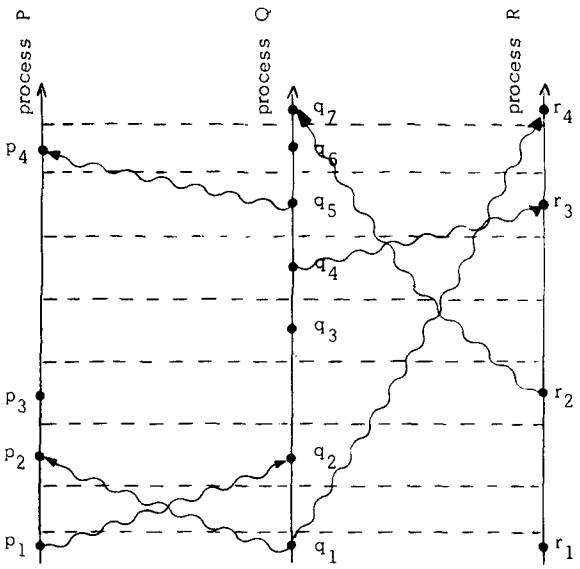
Logical Clocks

We now introduce clocks into the system. We begin with an abstract point of view in which a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. More precisely, we define a clock C_i for each process P_i to be a function which assigns a number $C_i\langle a\rangle$ to any event a in that process. The entire system of clocks is represented by the function C which assigns to any event b the number $C\langle b\rangle$, where $C\langle b\rangle = C_j\langle b\rangle$ if b is an event in process P_j. For now, we make no assumption about the relation of the numbers $C_i\langle a\rangle$ to physical time, so we can think of the clocks C_i as logical rather than physical clocks. They may be implemented by counters with no actual timing mechanism.

[1] The choice of what constitutes an event affects the ordering of events in a process. For example, the receipt of a message might denote the setting of an interrupt bit in a computer, or the execution of a subprogram to handle that interrupt. Since interrupts need not be handled in the order that they occur, this choice will affect the ordering of a process' message-receiving events.

[2] Observe that messages may be received out of order. We allow the sending of several messages to be a single event, but for convenience we will assume that the receipt of a single message does not coincide with the sending or receipt of any other message.

Fig. 3.



We now consider what it means for such a system of clocks to be correct. We cannot base our definition of correctness on physical time, since that would require introducing clocks which keep physical time. Our definition must be based on the order in which events occur. The strongest reasonable condition is that if an event a occurs before another event b, then a should happen at an earlier time than b. We state this condition more formally as follows.

Clock Condition. For any events a, b:

$$\text{if } a \rightarrow b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

Note that we cannot expect the converse condition to hold as well, since that would imply that any two concurrent events must occur at the same time. In Figure 1, p_2 and p_3 are both concurrent with q_3, so this would mean that they both must occur at the same time as q_3, which would contradict the Clock Condition because $p_2 \rightarrow p_3$.

It is easy to see from our definition of the relation "$\rightarrow$" that the Clock Condition is satisfied if the following two conditions hold.

C1. If a and b are events in process P_i, and a comes before b, then $C_i\langle a\rangle < C_i\langle b\rangle$.

C2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$.

Let us consider the clocks in terms of a space-time diagram. We imagine that a process' clock "ticks" through every number, with the ticks occurring between the process' events. For example, if a and b are consecutive events in process P_i with $C_i\langle a\rangle = 4$ and $C_i\langle b\rangle = 7$, then clock ticks 5, 6, and 7 occur between the two events. We draw a dashed "tick line" through all the like-numbered ticks of the different processes. The space-time diagram of Figure 1 might then yield the picture in Figure 2. Condition C1 means that there must be a tick line between any two events on a process line, and condition C2 means that every message line must cross a tick line. From the pictorial meaning of $\rightarrow$, it is easy to see why these two conditions imply the Clock Condition.

We can consider the tick lines to be the time coordinate lines of some Cartesian coordinate system on space-time. We can redraw Figure 2 to straighten these coordinate lines, thus obtaining Figure 3. Figure 3 is a valid alternate way of representing the same system of events as Figure 2. Without introducing the concept of physical time into the system (which requires introducing physical clocks), there is no way to decide which of these pictures is a better representation.

The reader may find it helpful to visualize a two-dimensional spatial network of processes, which yields a three-dimensional space-time diagram. Processes and messages are still represented by lines, but tick lines become two-dimensional surfaces.

Let us now assume that the processes are algorithms, and the events represent certain actions during their execution. We will show how to introduce clocks into the processes which satisfy the Clock Condition. Process P_i's clock is represented by a register C_i, so that $C_i\langle a\rangle$ is the value contained by C_i during the event a. The value of C_i will change between events, so changing C_i does not itself constitute an event.

To guarantee that the system of clocks satisfies the Clock Condition, we will insure that it satisfies conditions C1 and C2. Condition C1 is simple; the processes need only obey the following implementation rule:

IR1. Each process P_i increments C_i between any two successive events.

To meet condition C2, we require that each message m contain a *timestamp* T_m which equals the time at which the message was sent. Upon receiving a message time-stamped T_m, a process must advance its clock to be later than T_m. More precisely, we have the following rule.

IR2. (a) If event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i\langle a\rangle$. (b) Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

In IR2(b) we consider the event which represents the receipt of the message m to occur after the setting of C_j. (This is just a notational nuisance, and is irrelevant in any actual implementation.) Obviously, IR2 insures that C2 is satisfied. Hence, the simple implementation rules IR1 and IR2 imply that the Clock Condition is satisfied, so they guarantee a correct system of logical clocks.

Ordering the Events Totally

We can use a system of clocks satisfying the Clock Condition to place a total ordering on the set of all system events. We simply order the events by the times

at which they occur. To break ties, we use any arbitrary total ordering $\prec$ of the processes. More precisely, we define a relation $\Rightarrow$ as follows: if a is an event in process P_i and b is an event in process P_j, then $a \Rightarrow b$ if and only if either (i) $C_i\langle a\rangle < C_j\langle b\rangle$ or (ii) $C_i\langle a\rangle = C_j\langle b\rangle$ and $P_i \prec P_j$. It is easy to see that this defines a total ordering, and that the Clock Condition implies that if $a \rightarrow b$ then $a \Rightarrow b$. In other words, the relation $\Rightarrow$ is a way of completing the "happened before" partial ordering to a total ordering.[3]

The ordering $\Rightarrow$ depends upon the system of clocks C_i, and is not unique. Different choices of clocks which satisfy the Clock Condition yield different relations $\Rightarrow$. Given any total ordering relation $\Rightarrow$ which extends $\rightarrow$, there is a system of clocks satisfying the Clock Condition which yields that relation. It is only the partial ordering $\rightarrow$ which is uniquely determined by the system of events.

Being able to totally order the events can be very useful in implementing a distributed system. In fact, the reason for implementing a correct system of logical clocks is to obtain such a total ordering. We will illustrate the use of this total ordering of events by solving the following version of the mutual exclusion problem. Consider a system composed of a fixed collection of processes which share a single resource. Only one process can use the resource at a time, so the processes must synchronize themselves to avoid conflict. We wish to find an algorithm for granting the resource to a process which satisfies the following three conditions: (I) A process which has been granted the resource must release it before it can be granted to another process. (II) Different requests for the resource must be granted in the order in which they are made. (III) If every process which is granted the resource eventually releases it, then every request is eventually granted.

We assume that the resource is initially granted to exactly one process.

These are perfectly natural requirements. They precisely specify what it means for a solution to be correct.[4] Observe how the conditions involve the ordering of events. Condition II says nothing about which of two concurrently issued requests should be granted first.

It is important to realize that this is a nontrivial problem. Using a central scheduling process which grants requests in the order they are received will not work, unless additional assumptions are made. To see this, let P_0 be the scheduling process. Suppose P_1 sends a request to P_0 and then sends a message to P_2. Upon receiving the latter message, P_2 sends a request to P_0. It is possible for P_2's request to reach P_0 before P_1's request does. Condition II is then violated if P_2's request is granted first.

To solve the problem, we implement a system of clocks with rules IR1 and IR2, and use them to define a total ordering $\Rightarrow$ of all events. This provides a total ordering of all request and release operations. With this ordering, finding a solution becomes a straightforward exercise. It just involves making sure that each process learns about all other processes' operations.

To simplify the problem, we make some assumptions. They are not essential, but they are introduced to avoid distracting implementation details. We assume first of all that for any two processes P_i and P_j, the messages sent from P_i to P_j are received in the same order as they are sent. Moreover, we assume that every message is eventually received. (These assumptions can be avoided by introducing message numbers and message acknowledgment protocols.) We also assume that a process can send messages directly to every other process.

Each process maintains its own *request queue* which is never seen by any other process. We assume that the request queues initially contain the single message T_0:P_0 *requests resource*, where P_0 is the process initially granted the resource and T_0 is less than the initial value of any clock.

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process P_i sends the message T_m:P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.

2. When process P_j receives the message T_m:P_i *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.[5]

3. To release the resource, process P_i removes any T_m:P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.

4. When process P_j receives a P_i *releases resource* message, it removes any T_m:P_i *requests resource* message from its request queue.

5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a T_m:P_i *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $\Rightarrow$. (To define the relation "$\Rightarrow$" for messages, we identify a message with the event of sending it.) (ii) P_i has received a message from every other process timestamped later than T_m.[6]

Note that conditions (i) and (ii) of rule 5 are tested locally by P_i.

It is easy to verify that the algorithm defined by these rules satisfies conditions I–III. First of all, observe that condition (ii) of rule 5, together with the assumption that messages are received in order, guarantees that P_i has learned about all requests which preceded its current

[3] The ordering $\prec$ establishes a priority among the processes. If a "fairer" method is desired, then $\prec$ can be made a function of the clock value. For example, if $C_i\langle a\rangle = C_j\langle b\rangle$ and $j < i$, then we can let $a \Rightarrow b$ if $j < C_i\langle a\rangle \bmod N \leq i$, and $b \Rightarrow a$ otherwise; where N is the total number of processes.

[4] The term "eventually" should be made precise, but that would require too long a diversion from our main topic.

[5] This acknowledgment message need not be sent if P_j has already sent a message to P_i timestamped later than T_m.

[6] If $P_i \prec P_j$, then P_i need only have received a message timestamped $\geq T_m$ from P_j.

request. Since rules 3 and 4 are the only ones which delete messages from the request queue, it is then easy to see that condition I holds. Condition II follows from the fact that the total ordering $\Rightarrow$ extends the partial ordering $\rightarrow$. Rule 2 guarantees that after P_i requests the resource, condition (ii) of rule 5 will eventually hold. Rules 3 and 4 imply that if each process which is granted the resource eventually releases it, then condition (i) of rule 5 will eventually hold, thus proving condition III.

This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a *State Machine*, consisting of a set $\boldsymbol{C}$ of possible commands, a set $\boldsymbol{S}$ of possible states, and a function $\boldsymbol{e}: \boldsymbol{C} \times \boldsymbol{S} \rightarrow \boldsymbol{S}$. The relation $\boldsymbol{e}(C, S) = S'$ means that executing the command C with the machine in state S causes the machine state to change to S'. In our example, the set $\boldsymbol{C}$ consists of all the commands P_i *requests resource* and P_i *releases resource*, and the state consists of a queue of waiting *request* commands, where the request at the head of the queue is the currently granted one. Executing a *request* command adds the request to the tail of the queue, and executing a *release* command removes a command from the queue.[7]

Each process independently simulates the execution of the State Machine, using the commands issued by all the processes. Synchronization is achieved because all processes order the commands according to their timestamps (using the relation $\Rightarrow$), so each process uses the same sequence of commands. A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T. The precise algorithm is straightforward, and we will not bother to describe it.

This method allows one to implement any desired form of multiprocess synchronization in a distributed system. However, the resulting algorithm requires the active participation of all the processes. A process must know all the commands issued by other processes, so that the failure of a single process will make it impossible for any other process to execute State Machine commands, thereby halting the system.

The problem of failure is a difficult one, and it is beyond the scope of this paper to discuss it in any detail. We will just observe that the entire concept of failure is only meaningful in the context of physical time. Without physical time, there is no way to distinguish a failed process from one which is just pausing between events. A user can tell that a system has "crashed" only because he has been waiting too long for a response. A method which works despite the failure of individual processes or communication lines is described in [3].

[7] If each process does not strictly alternate *request* and *release* commands, then executing a *release* command could delete zero, one, or more than one request from the queue.

Anomalous Behavior

Our resource scheduling algorithm ordered the requests according to the total ordering $\Rightarrow$. This permits the following type of "anomalous behavior." Consider a nationwide system of interconnected computers. Suppose a person issues a request A on a computer A, and then telephones a friend in another city to have him issue a request B on a different computer B. It is quite possible for request B to receive a lower timestamp and be ordered before request A. This can happen because the system has no way of knowing that A actually preceded B, since that precedence information is based on messages external to the system.

Let us examine the source of the problem more closely. Let $\mathscr{S}$ be the set of all system events. Let us introduce a set of events which contains the events in $\mathscr{S}$ together with all other relevant external events, such as the phone calls in our example. Let $\rightarrow$ denote the "happened before" relation for $\underline{\mathscr{S}}$. In our example, we had A $\rightarrow$ B, but A $\not\rightarrow$ B. It is obvious that no algorithm based entirely upon events in $\mathscr{S}$, and which does not relate those events in any way with the other events in $\underline{\mathscr{S}}$, can guarantee that request A is ordered before request B.

There are two possible ways to avoid such anomalous behavior. The first way is to explicitly introduce into the system the necessary information about the ordering $\rightarrow$. In our example, the person issuing request A could receive the timestamp T_A of that request from the system. When issuing request B, his friend could specify that B be given a timestamp later than T_A. This gives the user the responsibility for avoiding anomalous behavior.

The second approach is to construct a system of clocks which satisfies the following condition.

Strong Clock Condition. For any events a, b in $\mathscr{S}$:
if $a \rightarrow b$ then $C\langle a\rangle < C\langle b\rangle$.

This is stronger than the ordinary Clock Condition because $\rightarrow$ is a stronger relation than $\rightarrow$. It is not in general satisfied by our logical clocks.

Let us identify $\underline{\mathscr{S}}$ with some set of "real" events in physical space-time, and let $\rightarrow$ be the partial ordering of events defined by special relativity. One of the mysteries of the universe is that it is possible to construct a system of physical clocks which, running quite independently of one another, will satisfy the Strong Clock Condition. We can therefore use physical clocks to eliminate anomalous behavior. We now turn our attention to such clocks.

Physical Clocks

Let us introduce a physical time coordinate into our space-time picture, and let $C_i(t)$ denote the reading of the clock C_i at physical time t.[8] For mathematical con-

[8] We will assume a Newtonian space-time. If the relative motion of the clocks or gravitational effects are not negligible, then $C_i(t)$ must be deduced from the actual clock reading by transforming from proper time to the arbitrarily chosen time coordinate.

venience, we assume that the clocks run continuously rather than in discrete "ticks." (A discrete clock can be thought of as a continuous one in which there is an error of up to $\frac{1}{2}$ "tick" in reading it.) More precisely, we assume that $C_i(t)$ is a continuous, differentiable function of t except for isolated jump discontinuities where the clock is reset. Then $dC_i(t)/dt$ represents the rate at which the clock is running at time t.

In order for the clock C_i to be a true physical clock, it must run at approximately the correct rate. That is, we must have $dC_i(t)/dt \approx 1$ for all t. More precisely, we will assume that the following condition is satisfied:

PC1. There exists a constant $\kappa \ll 1$ such that for all i: $|dC_i(t)/dt - 1| < \kappa$.

For typical crystal controlled clocks, $\kappa \leq 10^{-6}$.

It is not enough for the clocks individually to run at approximately the correct rate. They must be synchronized so that $C_i(t) \approx C_j(t)$ for all i, j, and t. More precisely, there must be a sufficiently small constant ϵ so that the following condition holds:

PC2. For all i, j: $|C_i(t) - C_j(t)| < \epsilon$.

If we consider vertical distance in Figure 2 to represent physical time, then PC2 states that the variation in height of a single tick line is less than ϵ.

Since two different clocks will never run at exactly the same rate, they will tend to drift further and further apart. We must therefore devise an algorithm to insure that PC2 always holds. First, however, let us examine how small κ and ϵ must be to prevent anomalous behavior. We must insure that the system $\underline{\mathcal{S}}$ of relevant physical events satisfies the Strong Clock Condition. We assume that our clocks satisfy the ordinary Clock Condition, so we need only require that the Strong Clock Condition holds when a and b are events in $\underline{\mathcal{S}}$ with $a \nrightarrow b$. Hence, we need only consider events occurring in different processes.

Let μ be a number such that if event a occurs at physical time t and event b in another process satisfies $a \rightarrow b$, then b occurs later than physical time $t + \mu$. In other words, μ is less than the shortest transmission time for interprocess messages. We can always choose μ equal to the shortest distance between processes divided by the speed of light. However, depending upon how messages in $\underline{\mathcal{S}}$ are transmitted, μ could be significantly larger.

To avoid anomalous behavior, we must make sure that for any i, j, and t: $C_i(t + \mu) - C_j(t) > 0$. Combining this with PC1 and 2 allows us to relate the required smallness of κ and ϵ to the value of μ as follows. We assume that when a clock is reset, it is always set forward and never back. (Setting it back could cause C1 to be violated.) PC1 then implies that $C_i(t + \mu) - C_i(t) > (1 - \kappa)\mu$. Using PC2, it is then easy to deduce that $C_i(t + \mu) - C_j(t) > 0$ if the following inequality holds:

$$\epsilon/(1 - \kappa) \leq \mu.$$

This inequality together with PC1 and PC2 implies that anomalous behavior is impossible.

We now describe our algorithm for insuring that PC2 holds. Let m be a message which is sent at physical time t and received at time t'. We define $\nu_m = t' - t$ to be the *total delay* of the message m. This delay will, of course, not be known to the process which receives m. However, we assume that the receiving process knows some *minimum delay* $\mu_m \geq 0$ such that $\mu_m \leq \nu_m$. We call $\xi_m = \nu_m - \mu_m$ the *unpredictable delay* of the message.

We now specialize rules IR1 and 2 for our physical clocks as follows:

IR1′. For each i, if P_i does not receive a message at physical time t, then C_i is differentiable at t and $dC_i(t)/dt > 0$.

IR2′. (a) If P_i sends a message m at physical time t, then m contains a timestamp $T_m = C_i(t)$. (b) Upon receiving a message m at time t', process P_j sets $C_j(t')$ equal to maximum $(C_j(t' - 0), T_m + \mu_m)$.[9]

Although the rules are formally specified in terms of the physical time parameter, a process only needs to know its own clock reading and the timestamps of messages it receives. For mathematical convenience, we are assuming that each event occurs at a precise instant of physical time, and different events in the same process occur at different times. These rules are then specializations of rules IR1 and IR2, so our system of clocks satisfies the Clock Condition. The fact that real events have a finite duration causes no difficulty in implementing the algorithm. The only real concern in the implementation is making sure that the discrete clock ticks are frequent enough so C1 is maintained.

We now show that this clock synchronizing algorithm can be used to satisfy condition PC2. We assume that the system of processes is described by a directed graph in which an arc from process P_i to process P_j represents a communication line over which messages are sent directly from P_i to P_j. We say that a message is sent over this arc every τ seconds if for any t, P_i sends at least one message to P_j between physical times t and $t + \tau$. The *diameter* of the directed graph is the smallest number d such that for any pair of distinct processes P_j, P_k, there is a path from P_j to P_k having at most d arcs.

In addition to establishing PC2, the following theorem bounds the length of time it can take the clocks to become synchronized when the system is first started.

THEOREM. Assume a strongly connected graph of processes with diameter d which always obeys rules IR1′ and IR2′. Assume that for any message m, $\mu_m \leq \mu$ for some constant μ, and that for all $t \geq t_0$: (a) PC1 holds. (b) There are constants τ and ξ such that every τ seconds a message with an unpredictable delay less than ξ is sent over every arc. Then PC2 is satisfied with $\epsilon \approx d(2\kappa\tau + \xi)$ for all $t \gtrsim t_0 + \tau d$, where the approximations assume $\mu + \xi \ll \tau$.

The proof of this theorem is surprisingly difficult, and is given in the Appendix. There has been a great deal of work done on the problem of synchronizing physical clocks. We refer the reader to [4] for an intro-

[9] $C_j(t' - 0) = \lim_{\delta \to 0} C_j(t' - |\delta|)$.

duction to the subject. The methods described in the literature are useful for estimating the message delays μ_m and for adjusting the clock frequencies dC_i/dt (for clocks which permit such an adjustment). However, the requirement that clocks are never set backwards seems to distinguish our situation from ones previously studied, and we believe this theorem to be a new result.

Conclusion

We have seen that the concept of "happening before" defines an invariant partial ordering of the events in a distributed multiprocess system. We described an algorithm for extending that partial ordering to a somewhat arbitrary total ordering, and showed how this total ordering can be used to solve a simple synchronization problem. A future paper will show how this approach can be extended to solve any synchronization problem.

The total ordering defined by the algorithm is somewhat arbitrary. It can produce anomalous behavior if it disagrees with the ordering perceived by the system's users. This can be prevented by the use of properly synchronized physical clocks. Our theorem showed how closely the clocks can be synchronized.

In a distributed system, it is important to realize that the order in which events occur is only a partial ordering. We believe that this idea is useful in understanding any multiprocess system. It should help one to understand the basic problems of multiprocessing independently of the mechanisms used to solve them.

Appendix

Proof of the Theorem

For any i and t, let us define C_i^t to be a clock which is set equal to C_i at time t and runs at the same rate as C_i, but is never reset. In other words,

$$C_i^t(t') = C_i(t) + \int_t^{t'} [dC_i(t)/dt]dt \tag{1}$$

for all $t' \geq t$. Note that

$$C_i(t') \geq C_i^t(t') \text{ for all } t' \geq t. \tag{2}$$

Suppose process P_1 at time t_1 sends a message to process P_2 which is received at time t_2 with an unpredictable delay $\leq \xi$, where $t_0 \leq t_1 \leq t_2$. Then for all $t \geq t_2$ we have:

$$\begin{aligned} C_2^{t_2}(t) &\geq C_2^{t_2}(t_2) + (1-\kappa)(t - t_2) && \text{[by (1) and PC1]} \\ &\geq C_1(t_1) + \mu_m + (1-\kappa)(t - t_2) && \text{[by IR2' (b)]} \\ &= C_1(t_1) + (1-\kappa)(t - t_1) - [(t_2 - t_1) - \mu_m] + \kappa(t_2 - t_1) \\ &\geq C_1(t_1) + (1-\kappa)(t - t_1) - \xi. \end{aligned}$$

Hence, with these assumptions, for all $t \geq t_2$ we have:

$$C_2^{t_2}(t) \geq C_1(t_1) + (1-\kappa)(t - t_1) - \xi. \tag{3}$$

Now suppose that for $i = 1, \ldots, n$ we have $t_i \leq t_i' < t_{i+1}$, $t_0 \leq t_1$, and that at time t_i' process P_i sends a message to process P_{i+1} which is received at time t_{i+1} with an unpredictable delay less than ξ. Then repeated application of the inequality (3) yields the following result for $t \geq t_{n+1}$.

$$C_{n+1}^{t_{n+1}}(t) \geq C_1(t_1') + (1-\kappa)(t - t_1') - n\xi. \tag{4}$$

From PC1, IR1′ and 2′ we deduce that

$$C_1(t_1') \geq C_1(t_1) + (1-\kappa)(t_1' - t_1).$$

Combining this with (4) and using (2), we get

$$C_{n+1}(t) \geq C_1(t_1) + (1-\kappa)(t - t_1) - n\xi \tag{5}$$

for $t \geq t_{n+1}$.

For any two processes P and P′, we can find a sequence of processes $P = P_0, P_1, \ldots, P_{n+1} = P'$, $n \leq d$, with communication arcs from each P_i to P_{i+1}. By hypothesis (b) we can find times t_i, t_i' with $t_i' - t_i \leq \tau$ and $t_{i+1} - t_i' \leq \nu$, where $\nu = \mu + \xi$. Hence, an inequality of the form (5) holds with $n \leq d$ whenever $t \geq t_1 + d(\tau + \nu)$. For any i, j and any t, t_1 with $t_1 \geq t_0$ and $t \geq t_1 + d(\tau + \nu)$ we therefore have:

$$C_i(t) \geq C_j(t_1) + (1-\kappa)(t - t_1) - d\xi. \tag{6}$$

Now let m be any message timestamped T_m, and suppose it is sent at time t and received at time t'. We pretend that m has a clock C_m which runs at a constant rate such that $C_m(t) = t_m$ and $C_m(t') = t_m + \mu_m$. Then $\mu_m \leq t' - t$ implies that $dC_m/dt \leq 1$. Rule IR2′ (b) simply sets $C_j(t')$ to *maximum* $(C_j(t' - 0), C_m(t'))$. Hence, clocks are reset only by setting them equal to other clocks.

For any time $t_x \geq t_0 + \mu/(1-\kappa)$, let C_x be the clock having the largest value at time t_x. Since all clocks run at a rate less than $1 + \kappa$, we have for all i and all $t \geq t_x$:

$$C_i(t) \leq C_x(t_x) + (1+\kappa)(t - t_x). \tag{7}$$

We now consider the following two cases: (i) C_x is the clock C_q of process P_q. (ii) C_x is the clock C_m of a message sent at time t_1 by process P_q. In case (i), (7) simply becomes

$$C_i(t) \leq C_q(t_x) + (1+\kappa)(t - t_x). \tag{8i}$$

In case (ii), since $C_m(t_1) = C_q(t_1)$ and $dC_m/dt \leq 1$, we have

$$C_x(t_x) \leq C_q(t_1) + (t_x - t_1).$$

Hence, (7) yields

$$C_i(t) \leq C_q(t_1) + (1+\kappa)(t - t_1). \tag{8ii}$$

Since $t_x \geq t_0 + \mu/(1-\kappa)$, we get

$$\begin{aligned} C_q(t_x - \mu/(1-\kappa)) &\leq C_q(t_x) - \mu && \text{[by PC1]} \\ &\leq C_m(t_x) - \mu && \text{[by choice of } m\text{]} \\ &\leq C_m(t_x) - (t_x - t_1)\mu_m/\nu_m && [\mu_m \leq \mu,\ t_x - t_1 \leq \nu_m] \\ &= T_m && \text{[by definition of } C_m\text{]} \\ &= C_q(t_1) && \text{[by IR2'(a)].} \end{aligned}$$

Hence, $C_q(t_x - \mu/(1-\kappa)) \leq C_q(t_1)$, so $t_x - t_1 \leq \mu/(1-\kappa)$ and thus $t_1 \geq t_0$.

Letting $t_1 = t_x$ in case (i), we can combine (8i) and (8ii) to deduce that for any t, t_x with $t \geq t_x \geq t_0 + \mu/(1 - \kappa)$ there is a process P_q and a time t_1 with $t_x - \mu/(1 - \kappa) \leq t_1 \leq t_x$ such that for all i:

$$C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1). \tag{9}$$

Choosing t and t_x with $t \geq t_x + d(\tau + \nu)$, we can combine (6) and (9) to conclude that there exists a t_1 and a process P_q such that for all i:

$$C_q(t_1) + (1 - \kappa)(t - t_1) - d\xi \leq C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1) \tag{10}$$

Letting $t = t_x + d(\tau + \nu)$, we get

$$d(\tau + \nu) \leq t - t_1 \leq d(\tau + \nu) + \mu/(1 - \kappa).$$

Combining this with (10), we get

$$C_q(t_1) + (t - t_1) - \kappa d(\tau + \nu) - d\xi \leq C_i(t) \leq C_q(t_1) + (t - t_1) + \kappa[d(\tau + \nu) + \mu/(1 - \kappa)] \tag{11}$$

Using the hypotheses that $\kappa \ll 1$ and $\mu \leq \nu \ll \tau$, we can rewrite (11) as the following approximate inequality.

$$C_q(t_1) + (t - t_1) - d(\kappa\tau + \xi) \lesssim C_i(t) \lesssim C_q(t_1) + (t - t_1) + d\kappa\tau. \tag{12}$$

Since this holds for all i, we get

$$|C_i(t) - C_j(t)| \lesssim d(2\kappa\tau + \xi),$$

and this holds for all $t \gtrsim t_0 + d\tau$. □

Note that relation (11) of the proof yields an exact upper bound for $|C_i(t) - C_j(t)|$ in case the assumption $\mu + \xi \ll \tau$ is invalid. An examination of the proof suggests a simple method for rapidly initializing the clocks, or resynchronizing them if they should go out of synchrony for any reason. Each process sends a message which is relayed to every other process. The procedure can be initiated by any process, and requires less than $2d(\mu + \xi)$ seconds to effect the synchronization, assuming each of the messages has an unpredictable delay less than ξ.

Acknowledgment. The use of timestamps to order operations, and the concept of anomalous behavior are due to Paul Johnson and Robert Thomas.

Received March 1976; revised October 1977

References

1. Schwartz, J.T. *Relativity in Illustrations.* New York U. Press, New York, 1962.
2. Taylor, E.F., and Wheeler, J.A. *Space-Time Physics*, W.H. Freeman, San Francisco, 1966.
3. Lamport, L. The implementation of reliable distributed multiprocess systems. To appear in *Computer Networks.*
4. Ellingson, C., and Kulpinski, R.J. Dissemination of system-time. *IEEE Trans. Comm. Com-23*, 5 (May 1973), 605–624.

Programming Languages | J. J. Horning Editor

Shallow Binding in Lisp 1.5

Henry G. Baker, Jr.
Massachusetts Institute of Technology

Shallow binding is a scheme which allows the value of a variable to be accessed in a bounded amount of computation. An elegant model for shallow binding in Lisp 1.5 is presented in which context-switching is an environment tree transformation called rerooting. Rerooting is completely general and reversible, and is optional in the sense that a Lisp 1.5 interpreter will operate correctly whether or not rerooting is invoked on every context change. Since rerooting leaves assoc [*v*, *a*] invariant, for all variables *v* and all environments *a*, the programmer can have access to a rerooting primitive, shallow[], which gives him dynamic control over whether accesses are shallow or deep, and which affects only the speed of execution of a program, not its semantics. In addition, multiple processes can be active in the same environment structure, so long as rerooting is an indivisible operation. Finally, the concept of rerooting is shown to combine the concept of shallow binding in Lisp with Dijkstra's display for Algol and hence is a general model for shallow binding.

Key Words and Phrases: Lisp 1.5, environment trees, FUNARG's, shallow binding, deep binding, multiprogramming, Algol display

CR Categories: 4.13, 4.22, 4.32

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0522.

Author's present address: Computer Science Department, University of Rochester, Rochester, NY 14627.