Timestamps in Message-Passing Systems That Preserve the Partial Ordering

Colin J. Fidge
Department of Computer Science, Australian National University, Canberra, ACT.

ABSTRACT

Timestamping is a common method of totally ordering events in concurrent programs. However, for applications requiring access to the global state, a total ordering is inappropriate. This paper presents algorithms for timestamping events in both synchronous and asynchronous message-passing programs that allow for access to the partial ordering inherent in a parallel system. The algorithms do not change the communications graph or require a central timestamp issuing authority.

Keywords and phrases: concurrent programming, message-passing, timestamps, logical clocks
CR categories: D.1.3

INTRODUCTION

A fundamental problem in concurrent programming is determining the order in which events in different processes occurred. An obvious solution is to attach a number representing the current time to a permanent record of the execution of each event. This assumes that each process can access an accurate clock, but practical parallel systems, by their very nature, make it difficult to ensure consistency among the processes.

There are two solutions to this problem. Firstly, have a central process to issue timestamps, i.e. provide the system with a global clock. In practice this has the major disadvantage of needing communication links from all processes to the central clock.

More acceptable are separate clocks in each process that are kept synchronised as much as necessary to ensure that the timestamps represent, at the very least, a *possible* ordering of events (in light of the vagaries of distributed scheduling). Lamport (1978) describes just such a scheme of logical clocks that can be used to totally order events, without the need to introduce extra communication links.

However this only yields one of the many possible, and equally valid, event orderings defined by a particular distributed computation. For problems concerned with the global program state it is far more useful to have access to the entire *partial* ordering, which defines the set of consistent "slices" of the global state at any arbitrary moment in time.

This paper presents an implementation of the partially ordered relation "happened before" that is true for two given events iff the first could causally affect the second in all possible interleavings of events. This allows access to *all* possible global states for a particular distributed computation, rather than a single, arbitrarily selected ordering. Lamport's totally ordered relation is used as a starting point. The algorithm is first defined for the asynchronous case, and then extended to cater for concurrent programs using synchronous message-passing.

A TOTAL ORDERING

For a system of parallel processes communicating via asynchronous signals, an arbitrary total ordering "$\Rightarrow$" can be placed on events as follows (Lamport, 1978).

Each process maintains an integer value, initially zero, which it periodically increments, e.g. once after every atomic event. This value is attached to the record of the execution of each event as its timestamp; for the purposes of this paper we will assume that the distributed system is recording, as it executes, a "history trace" of every event that executes. This may be done centrally, or separate traces may be maintained by each process.

Obviously these local logical clocks will quickly drift out of alignment. To overcome this the clocks are (roughly) synchronised by piggybacking the current local time onto every outgoing signal. Upon receiving a signal a process examines the attached clock value, and sets its own local clock to be greater than this value, if it is not already. This maintains consistency among the distributed clocks, since the departure of a signal is always timestamped as preceding its arrival (assuming that signals are the only form of communication between processes). See figure 1.

For two timestamped events a and b, $a \Rightarrow b$ iff the timestamp for a is less than that for b. Clearly some events in different processes may be assigned the same timestamp, in which case $a \not\Rightarrow b$ and $b \not\Rightarrow a$. The total ordering is completed by arbitrarily (but consistently) ordering the events in this case, for example, by assuming a fixed precedence between the different processes.

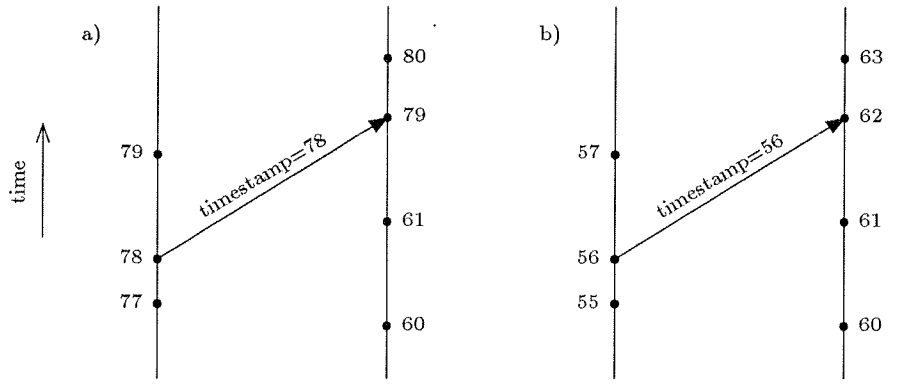

Figure 1. Lamport's algorithm: a) clock in sender running fast—clock in receiver advanced, b) clock in sender running slow—no special action required. Dots represent events (with their attached timestamps).

PROBLEM STATEMENT

Although always consistent with the observed behaviour of a distributed system, the $\Rightarrow$ relation only defines one of many possible event orderings for a given distributed computation. Knowledge of any other, equally valid, orderings is lost. Even the partial ordering resulting from some events having the same timestamp only preserves a small number of the potential orderings. Problems concerned with the global program state are best served by the opposite approach, retaining *all* potential orderings.

Consider the example in figure 2b. Here we wish to know that event n *could* occur concurrently with any of d, e or f. Similarly event f could occur concurrently with n, o or p. The desired relation is, of course, transitive. For example, event x could occur concurrently with any of a–f. If we are following the model that atomic events have no duration then "concurrently" is taken to mean that the events may occur in either order (alternatively, think of these "events" as locally saved process states—thus state n is consistent with states d, e and f).

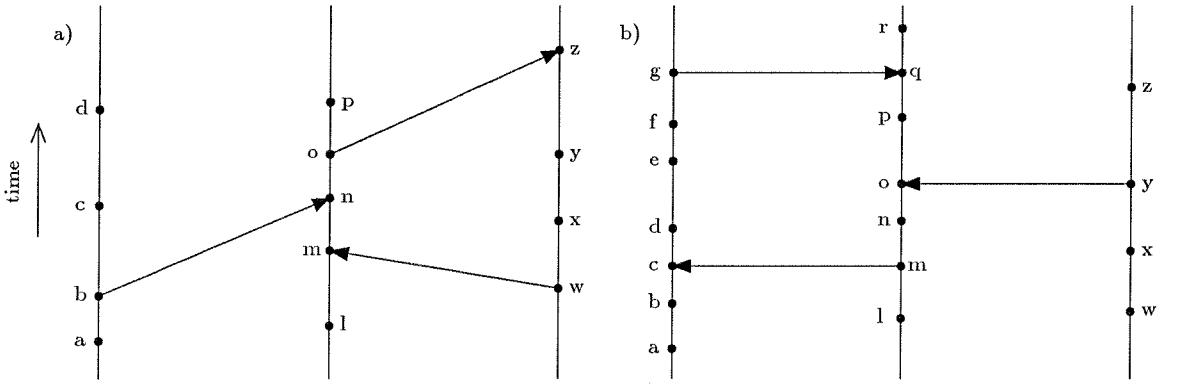

Figure 2. Example computations: a) asynchronous communication, b) synchronous communication.

By using simple integer values for timestamps, Lamport's algorithm cannot (and in fairness is not intended to) provide this type of information.

The aim therefore is to implement the relation "$\rightarrow$" ("happened before") in which $a \rightarrow b$ iff a must occur before b in all possible event interleavings for this particular distributed computation, i.e. a can causally affect b. If $a \not\rightarrow b$ and $b \not\rightarrow a$ then the two events may have occurred concurrently, i.e. they form part of a consistent slice of the global program state. This is denoted "$a \leftrightarrow b$". Lamport (1978) notes that $\rightarrow$ is irreflexive since it is not meaningful to say that an event happened before itself, i.e. $a \not\rightarrow a$.

THE PARTIAL ORDERING

The key to the algorithm is to recognise that communication events form "boundaries" that limit the possible interleavings of concurrent events. As an illustration, think of the events in figure 2 as beads that can slide up and down the process time lines. A line drawn horizontally through the diagram represents an arbitrary moment in time.

In the asynchronous case (figure 2a), event y could be pushed down to occur before l, or up to occur after p. Also event a could be pushed up to occur after m, but it could never occur at the same time as,

or after, event n. This is because the signal arrows must always point forwards in time, i.e. the departure of a signal must always precede its arrival. The signal from b to n precludes the possibility that a could have occurred after n, since b must strictly occur before n and a must occur before b.

Similarly for the synchronous case (figure 2b) event f could be pushed down to occur before n, or up to occur after p. However it could never occur after event r since the synchronous communication event g-q forms a barrier, temporally separating it from the time segment containing r.

Informally, therefore, $a \rightarrow b$ if there was a communication between the processes containing a and b that occurred after a and before b (the algorithms given below also allow for the case where either a, b or both are themselves communication events). It is therefore necessary for each process to know when it last communicated with every other process.

Asynchronous Communication

Rather than a single integer value, timestamps are represented as an array

$$[c_1, c_2 \ldots c_n]$$

with an integer clock value for every process in the network. Let

$$e_p$$

represent an event e executed by process p, and

$$T_{e_p}$$

the timestamp array attached to the permanent record of the execution of this event. For example the following timestamp

$$[4, 7, 6, 12]$$

attached to an event x in process 2 had local clock value

$$T_{x_2}[2] = 7$$

when x was executed, while the last known clock value for process 4 was

$$T_{x_2}[4] = 12$$

In practice the clock in process 4 may have advanced well beyond this value by the time this timestamp was recorded, but 12 is the most recent value available to process 2.

These timestamp arrays are maintained as follows:

Rule RA1: Initially all values are zero.

Rule RA2: The local clock value is incremented at least once before each atomic event.

Rule RA3: The current value of the entire timestamp array is piggybacked on every outgoing signal.

Rule RA4: Upon receiving a signal, a process sets the value of each entry in the timestamp array to be the maximum of the two corresponding values in the local array, and in the piggybacked array received. The value corresponding to the sender, however, is a special case and is set to be one greater than the value received (to allow for transit time), but only if the local value is not already greater than that received (to allow for signal "overtaking" as described below), i.e.

```
q?other_array; /* receive timestamp array from process q */
if local_array[q] ≤ other_array[q] then
    local_array[q] := 1 + other_array[q];
for i := 1 to n do
    local_array[i] := max(local_array[i], other_array[i]);
```

In this way each process receives updates about the clocks in other processes, including non-neighbours.

Rule RA5: Values in the timestamp arrays are never decremented.

For example see figure 3.

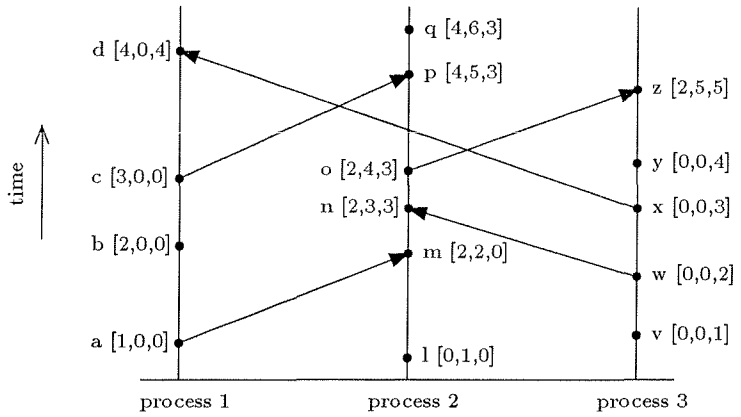

Figure 3. Use of timestamp arrays for asynchronous communication.

Timestamps attached to the stored records of events are compared as follows:

$$e_p \rightarrow f_q \quad \textbf{iff} \quad T_{e_p}[p] < T_{f_q}[p] \qquad \text{EA1}$$

In essence this says that event e_p must have occured before event f_q iff process q has received a signal from p (either directly or indirectly) that was sent after, or at the same time as, the execution of e_p.

Thus some of the temporal relationships defined by equation EA1 for two arbitrary events α and β in figure 3 are,

- α and β are different events in the same process:

$$\begin{array}{lll} w \rightarrow y & \text{since} & 2 < 4 \\ l \rightarrow p & & 1 < 5 \\ c \not\rightarrow b & & 3 \not< 2 \end{array}$$

- α and β are the same event:

$$\begin{array}{lll} c \not\rightarrow c & \text{since} & 3 \not< 3 \\ y \not\rightarrow y & & 4 \not< 4 \end{array}$$

  which implies that $c \leftrightarrow c$ and $y \leftrightarrow y$.

- α and β in different processes, no causal relationship:

$$\begin{array}{lll} l \leftrightarrow v & \text{since} & (1 \not< 0) \wedge (1 \not< 0) \\ d \leftrightarrow z & & (4 \not< 2) \wedge (5 \not< 4) \\ l \leftrightarrow b & & (2 \not< 0) \wedge (1 \not< 0) \end{array}$$

- α and β are events in different processes "separated" by an intervening communication:

$$\begin{array}{lll} b \rightarrow q & \text{since} & 2 < 4 \\ w \rightarrow n & & 2 < 3 \\ q \not\rightarrow c & & 6 \not< 0 \\ a \rightarrow z & & 1 < 2 \end{array}$$

Notice transitivity in this last case. Events a and z can be compared even though processes 1 and 3 have never directly communicated at this time.

The special action taken by the receiving process to compare the last known value of the sender, with the value just received (see rule RA4) is necessary to accomodate the possibility that signals do not always arrive in the order they are sent. In figure 4, for example, care must be taken to avoid setting the timestamp for event z to [7,4] since this would imply that $c \rightarrow z$. Some of the temporal relationships in figure 4 are:

$$\begin{array}{lll} a \rightarrow y & \text{since} & 4 < 6 \\ a \rightarrow z & & 4 < 6 \\ b \rightarrow y & & 5 < 6 \\ x \not\rightarrow c & & 2 \not< 0 \\ c \not\rightarrow x & & 6 \not< 0 \end{array}$$

The last two examples imply that $c \leftrightarrow x$.

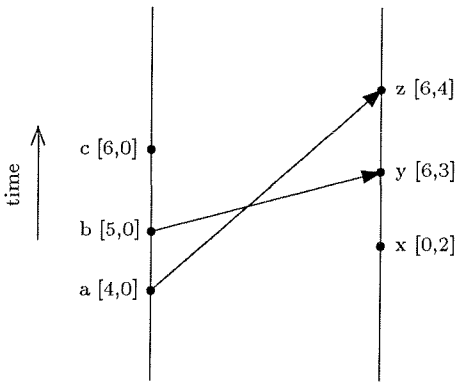

Figure 4. Example illustrating use of timestamp arrays during asynchronous signal "overtaking".

Appendix A justifies the asynchronous algorithm in more detail.

Synchronous Communication

If directly applied to a system using synchronous communication such as CSP (Hoare, 1978), Lamport's algorithm fails since it expects communication to take a finite amount of time, as illustrated by figure 5.

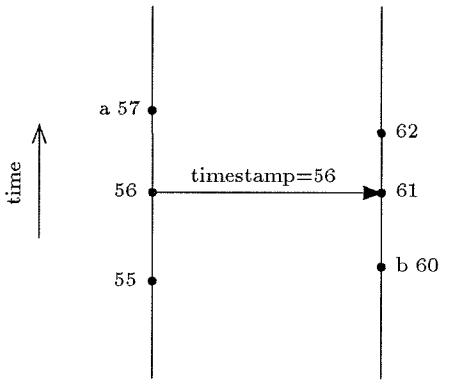

Figure 5. Failure of Lamport's algorithm when applied to synchronous communication. Event *a* is timestamped earlier than event *b* although *a must* occur after *b*.

This is easily solved by *exchanging* timestamps during the communication event, both processes setting their local clock to be equal to the largest value. This is necessary due to the symmetry of synchronous communication. It can be modelled as the receiving process sending back a dummy message with its local clock value to its communication partner, and both processes then adjusting their local clocks accordingly (figure 6). As long as all processes adhere to this protocol it cannot introduce deadlock.

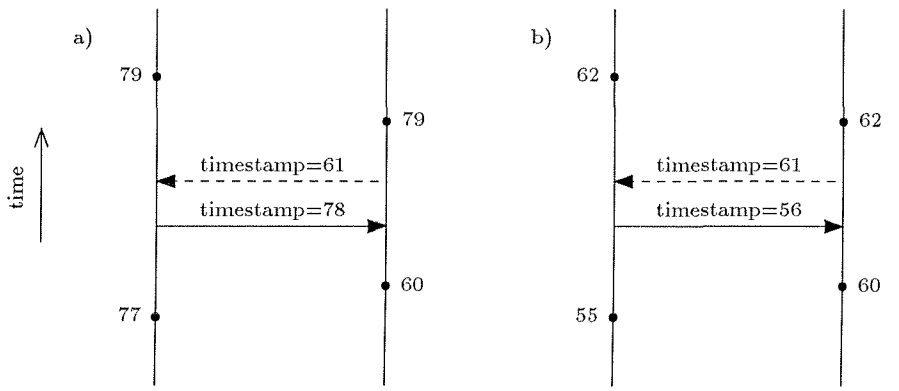

Figure 6. Lamport's algorithm adapted for synchronous communication: a) clock in sender running fast, b) clock in sender running slow.

For the remainder of this paper it is assumed that each synchronous communication is invisibly accompanied by this bi-directional timestamp exchange. Also note that due to the symmetry associated with this kind of communication, the direction of information transfer is unimportant. In future the directional arrows for synchronous communications will be omitted.

In the synchronous case the timestamp arrays are maintained as follows:

Rule RS1: Initially all values are zero.

Rule RS2: The local clock value is incremented at least once before each atomic event.

Rule RS3: During a communication event, the two processes involved exchange timestamp arrays and each element in the local array is set to be the maximum of its old value and the corresponding value in the array received, i.e.

```
q?other_array;
for i := 1 to n do
  local_array[i] := max(local_array[i], other_array[i]);
```

Rule RS4: Values in the timestamp arrays are never decremented.

See figure 7.

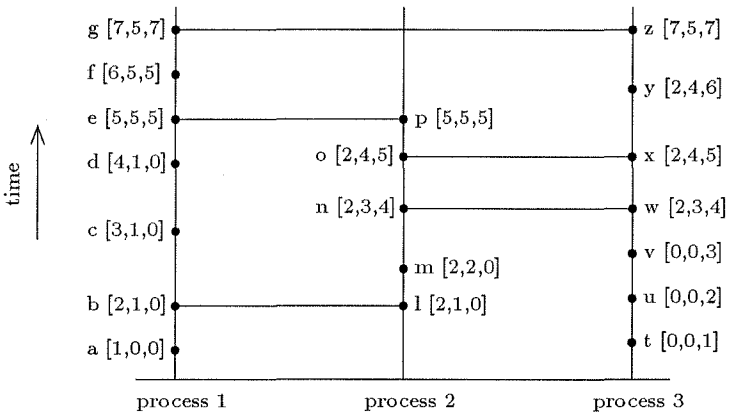

Figure 7. Use of timestamp arrays for synchronous communication. Each communication event appears twice to allow for a system in which each individual process separately records the execution of events in its own "history trace".

Notice the way that information is propogated around the network. For example, process 3 receives the time in process 1 at event w. In a global sense this value (2) is already out of date, but process 3's knowledge of process 1 is necessarily limited by the speed at which information can travel (assuming that synchronous message-passing is the only communications medium and that there are only the three processes shown in the network). Similarly process 1 learns that the clock in process 3 has reached (at least) 5 at event e, even though these two processes have not directly communicated at this point.

Also note that the timestamps for both processes executing a communication event are always the same since they both have up-to-date information about each other (the only time that this is ever possible). This conforms with the intuitive notion that a synchronous message-pass is a single, shared event.

Timestamps attached to the stored records of events are compared as follows:

$$e_p \rightarrow f_q \quad \textbf{iff} \quad T_{e_p}[p] \leq T_{f_q}[p] \ \wedge \ T_{e_p}[q] < T_{f_q}[q] \qquad \text{ES1}$$

This comparatively complex expression is explained as follows. The first half of the conjunction asserts that process q has received a clock value from process p at least as recent as the execution of event e_p. If this is the case then f_q *must* have been executed *after* e_p. The comparator is $\leq$ rather than $<$ to allow for the possibility that e_p was itself a communication event (in which case q *may* have up-to-date information about p when f_q was executed).

The second half of the conjunction simply states that process p does *not* have up-to-date information about process q, i.e. e_p is not the same event as f_p. This is necessary to avoid reflexivity. We have not

attempted to directly test "$e_p \neq f_q$" to allow for the possibility that each process is *independently* generating a history trace of timestamped events for post-mortem analysis, e.g. as in Fidge (1987), in which case it may not be clear which events are "matching halves" of a communication event. Alternatively we could simply have compared the entire timestamp arrays since they are always the same if $e_p \equiv f_q$, however this could become inefficient if the number of processes is large—the expression given above only needs to compare two integer values. Appendix B justifies equation ES1 in more detail.

Thus some of the temporal relationships between two arbitrary events α and β in figure 7 can be determined as follows,

- α and β in different processes, separated by a communication event:

$$\begin{array}{lll} a \rightarrow m & \text{since} & 1 \leq 2 \wedge 0 < 2 \\ v \rightarrow f & & 3 \leq 5 \wedge 0 < 6 \\ f \not\rightarrow v & & 6 \not\leq 0 \wedge 5 \not< 3 \end{array}$$

- α and β in different processes, α is a communication event:

$$\begin{array}{lll} b \rightarrow m & \text{since} & 2 \leq 2 \wedge 1 < 2 \\ e \not\rightarrow v & & 5 \not\leq 0 \wedge 5 \not< 3 \end{array}$$

- α and β are different events in the same process:

$$\begin{array}{lll} t \rightarrow v & \text{since} & 1 \leq 3 \wedge 1 < 3 \\ c \not\rightarrow a & & 3 \not\leq 1 \wedge 3 \not< 1 \end{array}$$

- α and β are the same atomic event, or different halves of the same communication event:

$$\begin{array}{lll} e \not\rightarrow p & \text{since} & 5 \leq 5 \wedge 5 \not< 5 \\ p \not\rightarrow e & & 5 \leq 5 \wedge 5 \not< 5 \\ y \not\rightarrow y & & 6 \leq 6 \wedge 6 \not< 6 \end{array}$$

  and therefore $e \leftrightarrow p$ and $y \leftrightarrow y$.

- α and β in different processes, no communication separating them:

$$\begin{array}{lll} c \leftrightarrow m & \text{since} & (3 \not\leq 2 \wedge 1 < 2) \wedge (2 \not\leq 1 \wedge 2 < 3) \\ d \leftrightarrow y & & (4 \not\leq 2 \wedge 0 < 6) \wedge (6 \not\leq 0 \wedge 2 < 4) \\ d \leftrightarrow t & & (4 \not\leq 0 \wedge 0 < 1) \wedge (1 \not\leq 0 \wedge 0 < 4) \end{array}$$

This last category is the most interesting, being the primary motivation for this work—it provides a method for detecting which events *may* occur concurrently. Considering the way that figure 7 is drawn it may seem surprising at first glance that event d is considered to be concurrent with two widely separated events t and y. However this is the entire purpose of the algorithm—to detect which events may *potentially* occur concurrently, in the absence of a global overview such as that provided by the diagram.

STATIC VS. DYNAMIC PROCESSES

For the purposes of this discussion we have assumed that there are a fixed number of processes allocated when the program begins execution, as in occam* (INMOS, 1984). However, provided that all processes can be *uniquely* identified, the algorithms can be trivially extended to handle dynamic process creation.

In this case the timestamp "arrays" should be thought of as extendable lists—each process adds a slot for a newly created process to the array when, and if, it receives any communication (either directly or indirectly) from that process. When comparing timestamps, if no entry exists for one of the processes involved we simply substitute zero. Otherwise the algorithms are unchanged.

* "occam" is a trademark of the Inmos Group of Companies.

NAMED CHANNELS VS. NAMED PROCESSES

To aid modular program development, the trend in *synchronous* message-passing language design is now oriented towards uniquely named channels (e.g. INMOS, 1984) rather than uniquely named processes (as in Hoare, 1978) for identifying i/o partners. However even if it is assumed that it is not possible to know the identity of the process on the other end of a particular channel, the algorithm is not affected. Each process merely needs to know its own place in the array—rule RS3 treats all slots in the array uniformly and there is thus no need to know the identity of the i/o partner during synchronous communication.

APPLICATIONS

The original motivation for this work was an effort to generalise the system presented in Baiardi *et al* (1986) for dynamically checking assertions in CSP programs. Here the assertions could only be expressed in terms of inter-process communications. The aim was to allow these expressions to also refer to internal process states and thus avoid any inter/intra-process distinction. This provides a more natural interface for applications where the user needs to place some sort of integrity constraint on the global program state. To achieve this state updates for the variables in the assertion expression were periodically sent to a monitor process which then evaluated the expression. However, since these updates may be received from several different processes at any time, it was difficult to know which values may validly be compared. The $\rightarrow$ relation provides a simple way of doing this.

In a more general sense, the implementation of the relation developed here allows timestamps to be attached to state snapshots saved independently by several processes, making it possible to easily check which states form a valid, consistent slice of the global program state. This is clearly a useful capability for problems in distributed systems such as reverse execution, error recovery or "rollback" (e.g. Kant and Silberschatz, 1985), restarting programs from stored local states, etc. For example, Fidge (1987) presents an algorithm for reproducible testing of CSP programs, by recording history traces for each process. To avoid the need to save extremely long traces, or the necessity of always replaying the entire test, the global state is periodically saved; each process saves its local state, consistency being maintained by co-ordinating the saves with special "marker" messages. However to improve efficiency redundant state saves could be avoided by including a timestamp from the process initiating a save in the marker message. Each process would then only save its local state if the last state saved is out-of-date according to this timestamp. Note that in the asynchronous case, any discussion of snapshots of the state of a program must consider signals in transit. Assuming that these are held in buffers in the sending or receiving processes, these buffers must be included in the stored state (along with values of local variables and the program counter) for the process.

Similarly the algorithms address problems such as that described by Sinha (1986): to allow for greater efficiency in scheduling requests to access data in a distributed database, requests are given a range of timestamps (amounting to a timeout period) so that they may be rescheduled by the server. The timestamped ordering may need to be reversed, i.e. it is made commutable. By not placing any ordering on those events whose interleaving is not important a variant of the algorithms described here could circumvent this difficulty entirely.

CONCLUSION

This paper has described algorithms for timestamping events in a distributed system that *only* define an order between two events when their temporal relationship is unambiguously defined by inter-process communications. Variants for both synchronous and asynchronous communication have been presented. They are general in the sense that they allow any event to be compared to any other, even if they are both in the same process or both in non-neighbouring processes. All processes are treated uniformly; there is no need to know anything about the connectivity of the parallel network.

The algorithms are considerably more complicated than Lamport's algorithm for obtaining an *arbitrary* total ordering, however this level of complexity seems to be inherent in any attempt to access the global state of a distributed system. Nevertheless they retain the desirable properties of not changing the communications graph and not introducing any extra communication events (assuming in the synchronous case that the timestamp "exchange" is built in to the message-passing implementation).

As a final point we go full-circle and note that a total ordering equivalent to the $\Rightarrow$ relation can be derived from the timestamp arrays if so desired, by replacing equation EA1 or ES1.

ACKNOWLEDGEMENTS

I would like to thank John Ophel for his careful review of a draft of this paper.

REFERENCES

BAIARDI, F., DE FRANCESCO, N. and VAGLINI, G. (1986): Development of a Debugger for a Concurrent Language, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, pp. 547–553.

FIDGE, C.J. (1987): Reproducible Tests in CSP, *The Australian Computer Journal*, Vol. 19, No. 2, pp. 92–98.

HOARE, C.A.R. (1978): Communicating Sequential Processes, *Communications of the ACM*, Vol. 21, No. 8, pp. 666–677.

INMOS LIMITED (1984): *occam Programming Manual*, Prentice-Hall.

KANT, K. and SILBERSCHATZ, A. (1985): Error Propagation and Recovery in Concurrent Environments, *The British Computer Journal*, Vol. 28, No. 5, pp. 466–473.

LAMPORT, L. (1978): Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565.

SINHA, M.K. (1986): Commutable Transactions and the Time_pad Synchronisation Mechanism for Distributed Systems, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 3, pp. 462–476.

APPENDIX A—Proof of Asynchronous Algorithm

The appendices seek to improve our confidence in the algorithms by showing that they correctly implement the desired relation.

Lamport (1978) defines $\rightarrow$ for the asynchronous case as:

> "the smallest relation satisfying the following three conditions: (i) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (ii) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (iii) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$."

We will examine the algorithm for each of these conditions.

(i) Let events a and b both belong to same process r, and a occurs before b. Assume

$$a_r \rightarrow b_r \tag{1}$$

then by definition (EA1)

$$T_{a_r}[r] < T_{b_r}[r] \tag{2}$$

which is trivially true due to rules RA2 and RA5.

(ii) Let a_r be a signal send and b_s be the reception of the same signal. Assume

$$a_r \rightarrow b_s \tag{3}$$

then by EA1

$$T_{a_r}[r] < T_{b_s}[r] \tag{4}$$

Now, assume that b_s^- is the event in process s immediately preceding b_s. Then by rule RA4

$$T_{a_r}[r] < (\textbf{if } T_{b_s^-}[r] \leq T_{a_r}[r] \textbf{ then } T_{a_r}[r] + 1 \textbf{ else } T_{b_s^-}[r]) \tag{5}$$

The expression on the RHS of (5) implies that

$$(T_{b_s^-}[r] \leq T_{a_r}[r] \ \wedge \ T_{a_r}[r] < T_{a_r}[r] + 1) \ \vee \ (T_{b_s^-}[r] > T_{a_r}[r] \ \wedge \ T_{b_s^-}[r] > T_{a_r}[r]) \tag{6}$$

which reduces to

$$(T_{b_s^-}[r] \leq T_{a_r}[r]) \ \vee \ (T_{b_s^-}[r] > T_{a_r}[r]) \tag{7}$$

which must be true.

(iii) Let events a, b and c belong to processes r, s and t respectively. Assume

$$a_r \rightarrow b_s \tag{8}$$

and

$$b_s \rightarrow c_t \tag{9}$$

Then by EA1, from (8)

$$T_{a_r}[r] < T_{b_s}[r] \tag{10}$$

Note that this is also true if r and s are the same process.

If s and t are the same process then (9) implies that

$$T_{b_s}[r] \leq T_{c_t}[r] \tag{11}$$

since values are monotonically increasing (rule RA5). Alternatively, if s and t are different processes, then equation (11) can still be derived from (9) and rules RA4 and RA5. Combining (10) and (11) gives

$$T_{a_r}[r] < T_{c_t}[r] \tag{12}$$

and by EA1

$$a_r \rightarrow c_t \tag{13}$$

as required.

It remains to be shown that the algorithm implements the "smallest" relation $\rightarrow$, i.e. there are no other cases where $\rightarrow$ holds. Informally this can be inferred from EA1 and the observation that $T_{f_q}[p]$ can *only* be incremented in two ways, by signal reception (rule RA4) or by the passage of time when both events are in the same process (rule RA2). In either case there is an unambiguous temporal ordering between the events. Any scenario in which we assume that $\rightarrow$ holds without any temporal relationship between the events will lead to a contradiction. For example, take the simplest case, reflexivity. If we assume

$$a_r \rightarrow a_r \tag{14}$$

then by EA1

$$T_{a_r}[r] < T_{a_r}[r] \tag{15}$$

which is immediately in error.

APPENDIX B—Proof of Synchronous Algorithm

The proof for the synchronous algorithm closely follows that given in appendix A. The definition of $\rightarrow$ is unchanged except for condition (ii):

(ii') If a is an output/input and b is the corresponding input/output and $a \rightarrow c$ then $b \rightarrow c$.

Notice that by definition a and b must belong to different processes in this case since a synchronous message-passing process cannot send messages to itself.

As in appendix A we now examine each of the three conditions defining the $\rightarrow$ relation in turn:

(i) Let a and b both belong to process r and a occurs before b. Assume

$$a_r \rightarrow b_r \tag{1}$$

then by ES1

$$T_{a_r}[r] \leq T_{b_r}[r] \ \wedge \ T_{a_r}[r] < T_{b_r}[r] \tag{2}$$

which reduces to

$$T_{a_r}[r] < T_{b_r}[r] \tag{3}$$

which must be true due to rules RS2 and RS4.

(ii') Let a_r be a communication event and b_s its corresponding "partner". Assume

$$a_r \rightarrow c_t \tag{4}$$

then by ES1

$$T_{a_r}[r] \leq T_{c_t}[r] \ \wedge \ T_{a_r}[t] < T_{c_t}[t] \tag{5}$$

By RS3 it is possible to see that

$$T_{a_r} = T_{b_s} \tag{6}$$

and hence

$$T_{a_r}[t] = T_{b_s}[t] \tag{7}$$

Thus from (5) and (7)

$$T_{b_s}[t] < T_{c_t}[t] \tag{8}$$

Rules RS3 and RS4 suggest the following general rule for any two events x_u and y_v where $x_u \rightarrow y_v$:

$$T_{x_u}[m] \leq T_{y_v}[m] \tag{9}$$

for any array element m, i.e. *all* elements in the arrays for two temporally related events are at least as great as that in the earlier array.

Therefore, from (4), (6) and (9)

$$T_{b_s}[s] \leq T_{c_t}[s] \tag{10}$$

and by ES1, equations (8) and (10) imply

$$b_s \rightarrow c_t \tag{11}$$

as required.

(iii) Let events a, b and c belong to processes r, s and t respectively. Assume

$$a_r \rightarrow b_s \tag{12}$$

and

$$b_s \rightarrow c_t \tag{13}$$

Then by ES1, from (12)

$$T_{a_r}[r] \leq T_{b_s}[r] \ \wedge \ T_{a_r}[s] < T_{b_s}[s] \tag{14}$$

Also, by ES1, from (13)

$$T_{b_s}[s] \leq T_{c_t}[s] \ \wedge \ T_{b_s}[t] < T_{c_t}[t] \tag{15}$$

From (12) and (9)

$$T_{a_r}[t] \leq T_{b_s}[t] \tag{16}$$

and thus from the second part of (15)

$$T_{a_r}[t] < T_{c_t}[t] \tag{17}$$

Also by (13) and (9)

$$T_{b_s}[r] \leq T_{c_t}[r] \tag{18}$$

and adding the first part of (14) gives

$$T_{a_r}[r] \leq T_{c_t}[r] \tag{19}$$

Together (17) and (19) imply

$$a_r \rightarrow c_t \tag{20}$$

by definition (ES1), as required.

Again, to be thorough, we should show that there are no other conditions that allow $\rightarrow$ to hold. This can be done using reasoning similar to that in appendix A.