Time and Clocks in Distributed Systems

Vijay K. Garg

TR ECE-PDS-1995-007 March 1995

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Time and Clocks in Distributed Systems

Vijay K. Garg*
Parallel and Distributed Systems Laboratory
email: garg@ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

March 7, 1995

Abstract

We describe implementation and properties of clocks in distributed systems. These clocks are useful for keeping information about causality rather than physical time. Depending on the level of information required, logical clocks, vector clocks, direct dependency clocks or matrix clocks may be useful. This paper is intended to be a tutorial on clocks in distributed systems. It not only provides algorithms for implementing clocks, but also a proof technique for showing properties of clock-like algorithms in a distributed system.

1 Introduction

The concept of time pervades our thinking. We use time in many ways. One important use of time is in providing ordering of events. We say that event α *happened before* event β if the physical time the event α happened, t_α is less than t_β, the time at which β happened. The use of this ordering requires access to some clocking mechanism which can record t_α and t_β. In absence of precisely synchronized clocks in distributed systems, this is an impossible task. This leads us to consider ordering relationships which can are useful and implementable in distributed systems.

Many applications require us to identify cause and effect relationships. Observe that if e is a cause of f, then e must happen before f in physical time. Thus, *causally precedes* relation is a subset of *happened before* relation. We will construct mechanisms in distributed systems that give us information about *causally precedes* relation.

Although it appears that, we use *causally precedes* relation instead of *happened before* because of our inability to get precisely synchronized clocks, this is not completely true. It turns out, fortunately, that *causally precedes* is more useful relation than *happened before*. One important reason is that we are usually interested in reasoning about correctness of a distributed program independent of relative processor execution speeds. This implies that if event e happened before f in one execution, then unless e causes f, f may occur before e in a different execution. Thus,

*supported in part by the NSF Grant CCR-9110605, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

causally precedes relation is independent of execution speed, whereas happened before is not. This aspect will be made more precise later in this paper.

At this point we note that we have focused on a single use of clocks - providing ordering of events. Real clocks serve other purposes. For example, a real clock can be used to determine the duration of any activity. This helps us in asserting statements such as "activity A takes more time than activities B and C combined". Such assertions are useful in constructing real-time programs, but will not concern us.

This paper is organized as follows. Section 2 describes a model of distributed system (or more accuratley, an execution of a distributed program) and presents some definitions which are used later. Section 3 describes a scalar clock called logical clock. Section 4 describes a vector clock. Section 5 describes an intermediate clock called direct dependency clock. Section 6 describes higher dimensional clocks.

To keep exposition simple, we have avoided citations in the main body of the paper. All citations have been grouped together at the end in Section 7 which contains bibliographic notes.

2 Model of a distributed system

2.1 Deposet

We assume a loosely-coupled message-passing system without any shared memory or a global clock. A distributed program consists of a set of n processes denoted by $\{P_1, P_2, ..., P_n\}$ communicating solely via asynchronous messages. We do not make any assumptions on the ordering or reliability of messages.

At the most abstract level, execution of a distributed system can be defined as a set of events. Some examples of events are - beginning of execution of the procedure *foo*, termination of the procedure *get*, sending of a message, receipt of a message etc. If all events happen at the same location (processor), then a natural ordering can be defined on this set. For example, if the sequence of events is $e.f.g.h$, then the ordering relation is $e < f < g < h$. In this example, given any two events such as e and g, either e occured before g, or g occured before e. Thus, there is a *total* order of events in a sequential system. One of the difficulties in distributed systems is that this relation is only a partial order.

Leslie Lamport was one of the first to recognize the importance of the ordering relation between events in a distributed system. He postulated that this relation (which he called happened-before) be irreflexive and transitive. We describe this relation in this section.

We will be concerned with a single run r of a distributed program. Each process P_i in that run generates an execution trace $s_{i,0}e_{i,0}s_{i,1} \ldots e_{i,l-1}s_{i,l}$, which is a finite sequence of local *states* and *events* in the process P_i. A state corresponds to the values of all variables in the process. The set of variables include program counter. Thus, barring internal non-determinism, the event executed next is completely determined from this state and external communication. For some applications, it is sufficient to describe traces using only states while for others modeling based on pure events is simpler.

A *run* r is a vector of traces with $r[i]$ as the trace of the process P_i. We define a relation *locally precedes* denoted by $\prec_{im}$ between states in the trace of a single process P_i as follows: $s \prec_{im} t$ if and only if s *immediately* precedes t in the trace $r[i]$. We also say that $s.next = t$ or $t.prev = s$ whenever $s \prec_{im} t$. We use $\prec$ for irreflexive transitive closure and $\preceq$ for for reflexive transitive closure of $\prec_{im}$. Similar definitions hold for events.

We say that event e in the trace $r[i] \rightsquigarrow$ event f in the trace $r[j]$ if e is the send of a message and f is the receive event of the same message. Similarly, states s and t in the traces $r[i]$ and $r[j]$ are defined to be related by $\rightsquigarrow$ if and only if a message is sent by P_i after the state s which is received by P_j resulting in the state t. Figure 1 illustrates a run. Such figures are usually referred to as space-time diagrams or process-time diagrams.

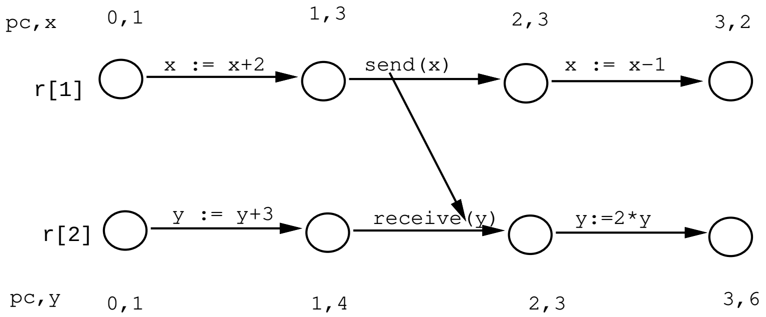


Figure 1: An example of a run

Now, the *causally precedes* relation can be defined as the transitive closure of union of $\prec_{im}$ and $\rightsquigarrow$. That is, $s \rightarrow t$ iff

1. $(s \prec_{im} t) \vee (s \rightsquigarrow t)$, or
2. $\exists u : (s \rightarrow u) \wedge (u \rightarrow t)$

In terms of process-time diagram, this means that there is a path from state s to state t. Intuitively, this relation captures *possible* cause-effect relationship between states. Causally precedes relation is a partial order on the set of local states.
We say that s and t are concurrent (denoted by $s||t$) if $\neg(s \rightarrow t) \ \wedge \ \neg(t \rightarrow s)$. We also use $s.p$ to denote the process in whose trace s occurs. That is, $s.p = i$ if and only if $s \in S_i$.

From now on, we will restrict ourselves to states generated during execution of the program. All definitions and results carry over to events without any difficulty.

Let S_i be the sequence of local states in P_i. With a slight abuse of notation, let $S = \bigcup_i S_i$. Then, an execution of a distributed program that consists of processes $P_1, \ldots P_N$ can be modeled with a tuple $(S_1, \ldots S_N, \rightsquigarrow)$. We call such a structure a deposet (decomposed partially ordered set) provided $(S, \rightarrow)$ is a partial order and it satisfies the following two natural conditions. First, it is assumed that there is no message received before the initial state or sent after the final state in any sequence S_i. Secondly, we assume that there is at most one message sent or received between any two consecutive states. To formalize, we define the initial and final states: $Init(i) \stackrel{\text{def}}{=} \min S_i$ and $Final(i) \stackrel{\text{def}}{=} \max S_i$.

Thus, a deposet can be defined as follows.

Definition 1 *A deposet is a tuple* $(S_1, \ldots S_N, \leadsto)$ *such that* $(S, \rightarrow)$ *as an irreflexive partial order which satisfies:*

1. *(D1)* $\forall i : \neg(\exists u :: u \leadsto Init(i))$
2. *(D2)* $\forall i : \neg(\exists u :: Final(i) \leadsto u)$
3. *(D3)* $s \prec t \Rightarrow \mid \{u \mid s \leadsto u \vee u \leadsto t\} \mid \leq 1$

We will use deposet as the formal model of execution of a distributed program.

2.2 Verification of Distributed Programs

Since we will later be specifying and verifying clock programs, we describe here our method of verification based on the deposet structure. The steps in our method are as follows:

1. The program is specified using the relation $s \prec_{im} t$ and $s \leadsto t$. Any poset that satisfies this relation is a valid computation of the program.
2. The desired properties of the program are specified using $\rightarrow, \not\rightarrow$ and other such relations. It is important to note that the concept of global time and therefore the global state is completely avoided in this approach. Any variable, say v, has meaning only in the context of the local state that it belongs to.
3. The desired properties are shown using properties derived from the program text.

For every pair of consecutive states, $s \prec_{im} t$, exactly one event occurs between s and t. There are three types of events denoted by int, snd, and rcv. Which event occurs between two states can be determined from the following definitions.

$$\begin{aligned}
(s, snd(u), t) &\stackrel{\triangle}{=} s \prec_{im} t \ \wedge \ s \leadsto u \\
(s, rcv(u), t) &\stackrel{\triangle}{=} s \prec_{im} t \ \wedge \ u \leadsto t \\
(s, snd, t) &\stackrel{\triangle}{=} (\exists u :: (s, snd(u), t)) \\
(s, rcv, t) &\stackrel{\triangle}{=} (\exists u :: (s, rcv(u), t)) \\
(s, int, t) &\stackrel{\triangle}{=} s \prec_{im} t \ \wedge \ \neg(s, snd, t) \ \wedge \ \neg(s, rcv, t)
\end{aligned}$$

The above relations model the events that occur between consecutive local states: (s, snd, t) models a message send, (s, rcv, t) models a message receive, and (s, int, t) models an internal event. We have assumed that the message delivery is reliable in the above model.

3 Logical Clock

A *logical clock* C is a map from S to $\mathcal{N}$ (the set of natural numbers) with the following constraint:

$$\forall s, t \in S : s \prec_{im} t \ \vee \ s \leadsto t \Rightarrow C(s) < C(t)$$

We use $\mathcal{C}$ to denote the set of all logical clocks which satisfy the above constraint. The interpretation of $C(s)$ for any $s \in S$ is that the process $s.p$ enters the state s when the clock value is $C(s)$. This constraint models the sequential nature of execution at each process and the physical requirement that any message transmission requires a non-zero amount of time. From the definition of $\rightarrow$, it is equivalent to

$$\forall s, t \in S : s \rightarrow t \Rightarrow \forall C \in \mathcal{C} : C(s) < C(t) \qquad \textbf{(CC)}$$

The condition (CC) is widely used as the definition of a logical clock since its proposal by Lamport.

We have not shown yet that $\mathcal{C}$, the set of all logical clocks, is a non-empty set. Since the run is assumed to be generated from a physical execution, by assigning the physical time to each state we derive a possible logical clock. The non-emptiness of $\mathcal{C}$ is equivalent to the condition that $\rightarrow$ is an irreflexive partial order. This is shown in the next Lemma.

Lemma 2 *$\mathcal{C}$ is non-empty iff $(S, \rightarrow)$ is an irreflexive partial order.*

Proof: ($\Rightarrow$) Let $D \in \mathcal{C}$. This implies that $\neg(s \rightarrow s)$, otherwise by (CC) $D(s) < D(s)$. Thus, $\rightarrow$ is irreflexive. By definition, $\rightarrow$ is transitive. It follows that $(S, \rightarrow)$ is an irreflexive partial order.
($\Leftarrow$) We are given that $(S, \rightarrow)$ is an irreflexive partial order. We define a map D from S to $\mathcal{N}$ as follows. $D(s)$ is the length of the longest sequence $s_0, s_1, ..., s_n$, where $s_0 = \bot_i$ for some i, $s_n = s$ and $s_k \rightarrow s_{k+1}$ for $0 \leq k \leq n-1$. It can be easily verified that D satisfies (CC).

■

From now on we will assume that $(S, \rightarrow)$ is a partial order, or equivalently that $\mathcal{C}$ is a non-empty set.

We now show that the set $\mathcal{C}$ also satisfies the converse of (CC), i.e.,

$$\forall s, t \in S : s \not\rightarrow t \Rightarrow \exists C \in \mathcal{C} : \neg(C(s) < C(t))$$

To this end, it is sufficient to show that

$$\forall u, v \in S : u || v \Rightarrow \exists C \in \mathcal{C} : (C(u) = C(v))$$

That is, if two local states are concurrent, then there exists a logical clock such that both states are assigned the same timestamp. We show this result for any subset X of S which contains pairwise concurrent states.

Lemma 3 *For any $X \subseteq S$,*
$(\forall u, v \in X : u || v) \Rightarrow \exists C \in \mathcal{C} : \forall u, v \in X : C(u) = C(v)$.

Proof: As $\mathcal{C}$ is a non-empty set, let $D \in \mathcal{C}$. D satisfies clock conditions, but may not assign the same value to all states in X. From D, we construct another logical clock C as follows:

(L1): For all $s \in X$: $\quad C(s) := max_{u \in X} D(u)$;
(L2): For all $s \in S - X$ such that $\neg \exists u \in X : u \rightarrow s$, we set $C(s) := D(s)$;
(L3): For all other s, we set $C(s) := D(s) + max_{u \rightarrow s}(C(u) - D(u))$

In other words, the new timestamp of all states in X is equal to the maximum of all the previous timestamps. States which do not depend on states in X retain their old timestamps. Other states have their timestamp increased by the amount given by (L3). For an illustration of this assignment see Figure 2.

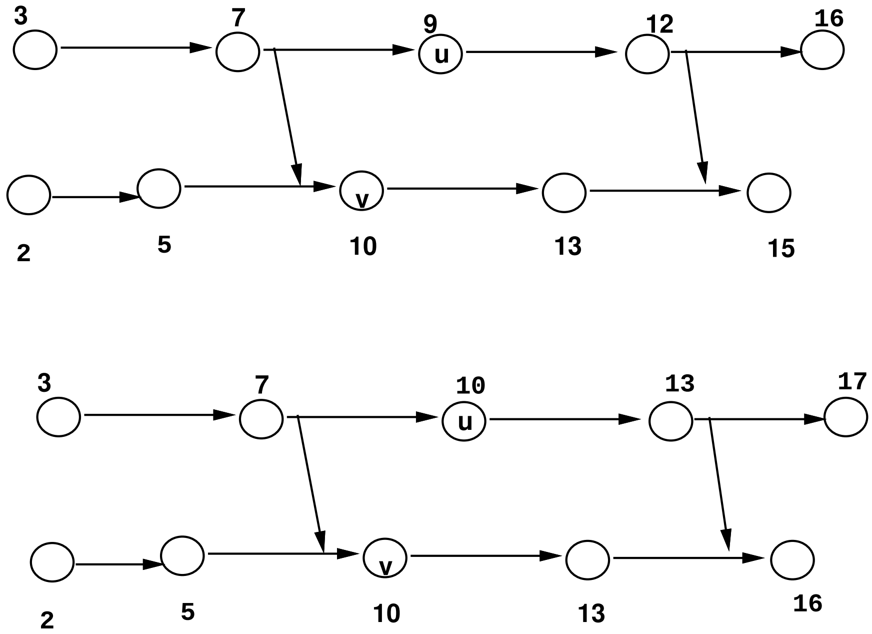


Figure 2: Illustration of clock assignment

From (L1), all states in X are assigned the same value by C. We need to show that C satisfies the condition (CC).
Since D satisfies (CC), $D(s) < D(t)$. Let $s \to t$. We now do case analysis.

Case 1: $s \in X$
Since X consists of pairwise concurrent states $t \notin X$. By (L3),

$$C(t) \geq D(t) + C(s) - D(s).$$

Since $D(s) < D(t)$, we get that $C(t) > C(s)$.

Case 2: $s \in S - X$ and $\neg \exists u \in X : u \to s$
By (L2), $C(s) = D(s)$. This implies that $C(s) < D(t)$, because $D(s) < D(t)$. Since D is uniformly less than C, we get that $C(s) < C(t)$.

Case 3: $s \in S - X$ and $\exists u \in X : u \to s$
By (L3),

$$C(s) = D(s) + max_{u \to s}(C(u) - D(u))$$

Since $D(s) < D(t)$, it follows that

$$C(s) < D(t) + max_{u \to s}(C(u) - D(u))$$

Further, $u \to s$ implies that $u \to t$. Therefore,

$$C(s) < D(t) + max_{u \to t}(C(u) - D(u))$$

Hence, $C(s) < C(t)$ by (L3).

Thus, C satisfies (CC).

■

Combining Lemma 3 with (CC), we get the following pleasant characterization of $\to$:

$$\forall s, t \in S : (s \to t \Leftrightarrow \forall C \in \mathcal{C} : C(s) < C(t))$$

Intuitively, the above formula says that s causally precedes t in a run r if and only if all possible observers of the run agree that s happened before t. Observe that *causally precedes* relation is a subset of any *happened before* relation observed by a single observer. Further, from the above characterization it is independent of the observation. This property of *causally precedes* relation makes it more useful than *happened before* relation based on real-time as we mentioned earlier.

At this point, it is useful to consider the distinction between *simultaneity* and *concurrency*. If different processes do not share a clock and the communication network takes uncertain amount of delay then it is impossible to determine simultaneity of two events. Therefore, in distributed systems the notion of concurrency is often used for possible simultaneity. Recall that two events e and f to be concurrent iff $e \not\to f$ and $f \not\to e$. Thus, two events are concurrent if neither can affect the other. It is important to observe that simultaneity and concurrency relations do not satisfy same properties. For example, simultaneity is a transitive relation whereas it is quite possible for $e \| f$, $f \| g$, but $e \not\| g$. It can be easily shown that

$$\forall s, t \in S : s \| t \Leftrightarrow \exists C \in \mathcal{C} : C(s) = C(t))$$

3.1 Implementation

Availability of a logical (or a global) clock during distributed computation makes solving many distributed problems easier. Recall that C is a logical clock iff it satisfies the following:

$$\forall s, t \in S : s \to t \Rightarrow C(s) < C(t)$$

An accurate physical clock clearly satisfies the above condition and therefore is also a logical clock. However, by definition of a distributed system there is no shared clock in the system. We now show an implementation of a logical clock which does not use any shared physical clock or shared memory.

Recall that for any state s, $s.p$ indicates the identity of the process to which it belongs. It is not required that message communication be ordered or reliable. The algorithm is described by the initial conditions and the actions taken for each event type. The algorithm uses the variable c to assign the logical clock. The notation $s.c$ is used to denote the value of c in the state s.

For any initial state s:
$integers.c = 0;$

Rule for a send event (s, snd, t):
/* s.c is sent as part of the message */
$t.c := s.c + 1;$

Rule for a receive event $(s, rcv(u), t)$:
$t.c := \max(s.c, u.c) + 1;$

Rule for an internal event (s, int, t):
$t.c := s.c + 1;$

The following claim is easy to verify:

$$\forall s, t \in S : s \rightarrow t \Rightarrow s.c < t.c$$

In some applications it is required that all events in the system be ordered totally. If we extend the logical clock with process number, then we get a total oredring on events. Thus, the timestamp of any event is a tuple $< e.c, e.p >$ and the total order $<$ is obtained as:
$(e.c, e.p) < (f.c, f.p)$ iff $(e.c < f.c) \vee ((e.c = f.c) \ \wedge \ (e.p < f.p))$.

4 Vector Clocks

4.1 Definition

We saw that logical clocks satisfy the following property:

$$s \rightarrow t \Rightarrow s.c < t.c.$$

However, the converse is not true. In this section, we describe a mechanism called vector clock which satisfies

$$s \rightarrow t \Leftrightarrow s.v < t.v.$$

We give an algorithm that assigns a vector to each local state such that the partial order is preserved. Vectors of integers can be partially ordered by an appropriately defined comparison relation $<$. Hence they are useful for characterizing the relationship between local states.

$$\begin{aligned} x < y &= (\forall k : 1 \leq k \leq N : x[k] \leq y[k]) \ \wedge \\ & \quad (\exists j : 1 \leq j \leq N : x[j] < y[j]) \\ x \leq y &= (x < y) \vee (x = y) \end{aligned}$$

Let there be N processes uniquely identified by an integer value between 1 and N inclusive. The algorithm is described by the initial conditions and the actions taken for each event type. An example is given in Fig. 3. It is not required that message communication be ordered or reliable.

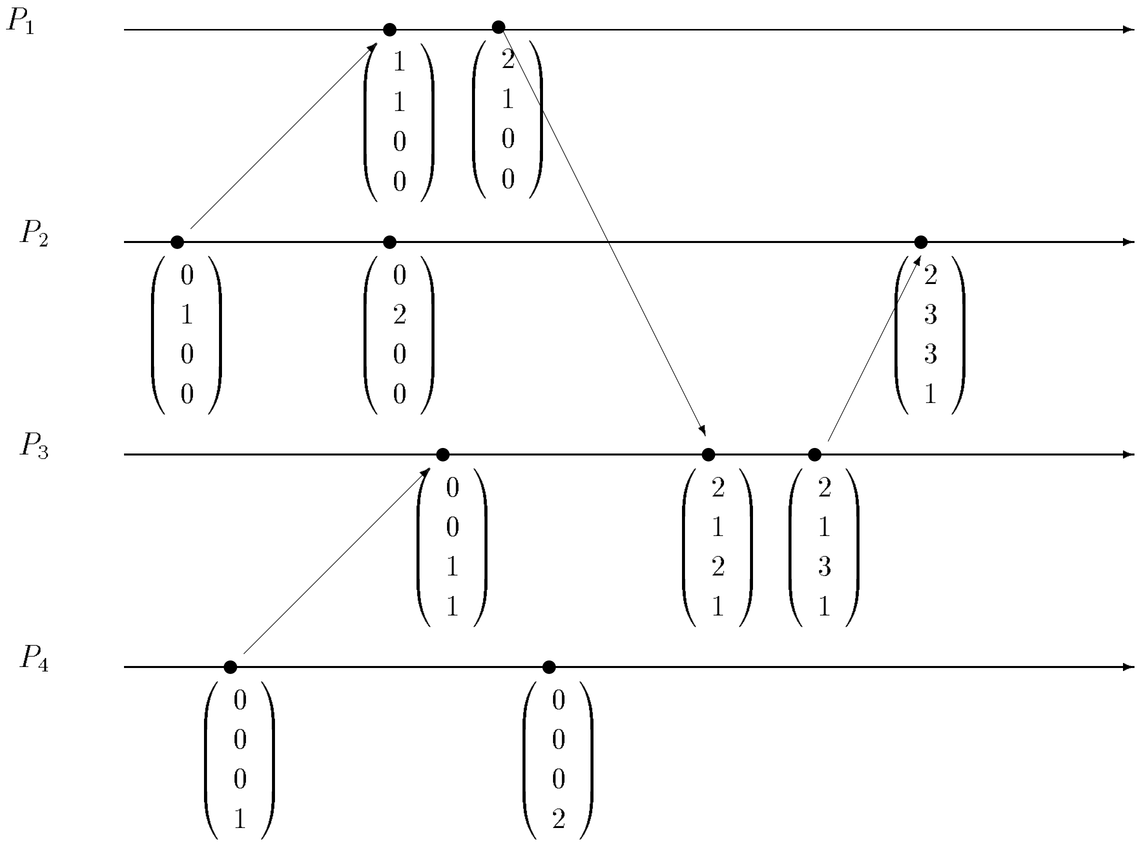


Figure 3: A sample execution of the vector clock algorithm

For any initial state s:
$(\forall i : i \neq s.p : s.v[i] = 0) \ \wedge \ (s.v[s.p] = 1)$

Rule for a send event (s, snd, t):
$t.v := s.v;$
$t.v[t.p] + +;$

Rule for a receive event $(s, rcv(u), t)$:
for $i := 1$ to N
$t.v[i] := \max(s.v[i], u.v[i]);$
$t.v[t.p] + +;$

Rule for an internal event (s, int, t):
$t.v := s.v;$
$t.v[t.p] + +;$

We show some properties of the above program informally. Rigorous proofs will be provided later.

Lemma 4 *Let $s \neq t$. Then,*

$$s \not\rightarrow t \Rightarrow t.v[s.p] < s.v[s.p]$$

Proof: If $t.p = s.p$, then it follows that $t \prec s$. Then, it is easy to see that $t.v[s.p] < s.v[s.p]$. So, we assume that $s.p \neq t.p$. Since $s.v[s.p]$ is the local clock of $P_{s.p}$ and $P_{t.p}$ could not have seen this value as $s \not\rightarrow t$, it follows that $t.v[s.p] < s.v[s.p]$.

■

Theorem 5 *Let s and t be states in processes P_i and P_j with vectors $s.v$ and $t.v$, respectively. Then, $s \rightarrow t$* **iff** *$s.v < t.v$.*

Proof: $(s \rightarrow t) \Rightarrow (s.v < t.v)$
If $s \rightarrow t$, then there is a message path from s to t. Therefore, since P_j updates its vector upon receipt of a message and this update is done by taking the component-wise maximum, we know the following holds: $\forall k : s.v[k] \leq t.v[k]$. Furthermore, since $t \not\rightarrow s$, from lemma 4
$t.v[j] > s.v[j]$. Hence, the following holds: $(\forall k : s.v[k] \leq t.v[k]) \wedge (s.v[j] < t.v[j])$ Thus, $(s \rightarrow t) \Rightarrow (s.v < t.v)$.

The converse follows from Lemma 4.

■

Recall that for any two vectors, x and y, $x < y$ if and only if $(\forall i : x[i] \leq y[i]) \wedge (\exists j : x[j] < y[j])$. Now we show that if we know the processes the vectors came from, the comparison between two states can be made in constant time.

Lemma 6 *$s \rightarrow t$ iff*

$$(s.v[s.p] \leq t.v[s.p]) \wedge (s.v[t.p] < t.v[t.p])$$

Proof: Left to the reader.

■

4.2 Using Induction to Prove Properties of Clocks

The causally-precedes relation, $\rightarrow$, and its complement, $\not\rightarrow$, has been quite useful in designing, analyzing and debugging asynchronous distributed programs. In this section, we define variants of these relations so that properties based on them can be proven by induction. The new relations defined in this section are $\stackrel{k}{\rightarrow}$ and $\stackrel{k}{\not\rightarrow}$. Figure 4 shows examples of these two relations.

A chain in $(S, \rightarrow)$ is a sequence of states $c_0, c_1, \ldots c_n$ such that $c_i \prec_{im} c_{i+1}$ or $c_i \rightsquigarrow c_{i+1}$. For any chain $c = c_0, c_1, \ldots c_n$, we define $first(c) = c_0$, $last(c) = c_n$, and $length(c) = n$. For any pair of states s, t we define the maximum length function, $ml(s,t)$ as follows:

$$(\max c : first(c) = s \ \wedge \ last(c) = t : length(c))$$

if $s \rightarrow t \ \vee \ s = t$ and -1 otherwise.

If $s \rightarrow t$, then $ml(s,t)$ equals the length of the longest chain between s and t. If $s = t$, then $ml(s,t) = 0$. We use $ml(Init, t)$ to denote $(\max u : Init(u) : ml(u,t))$. Thus $ml(Init, t)$ is length of the longest chain from some initial state to t. The following statement is true by definition of ml.

$$ml(s,t) > 0 \Leftrightarrow$$
$$(\exists u :: ml(s,u) = ml(s,t) - 1 \ \wedge \ ml(u,t) = 1)$$

We define the $\xrightarrow{k}$ relation which is used for induction on $\rightarrow$. For $k > 0$ we define

$$s \xrightarrow{k} t \triangleq ml(s,t) = k$$

Thus $s \xrightarrow{k} t$ if and only if $s \rightarrow t$ and the longest chain from s to t has length k.

In combination with the results of lemmas 7, 8 and 9, this new relation can be used to prove claims which are expressed in terms of $\rightarrow$. For example, suppose we wish to prove the claim $s \rightarrow t \Rightarrow P(s,t)$ where $P(s,t)$ is some predicate on the local variables in s and t. From lemma 7 we note that it is sufficient to prove $s \xrightarrow{k} t \Rightarrow P(s,t)$ for all $k > 0$. The proof can proceed by induction on k, using lemma 8 for the base case and lemma 9 for the induction. The base case, $s \xrightarrow{1} t$, implies that either states s and t are consecutive states in a process, or a message was sent in s and received in t. Generally the base case can be easily proven from the program text since it involves only one state transition or one message.

Lemma 7 $s \rightarrow t \Leftrightarrow (\exists k : k > 0 : s \xrightarrow{k} t)$

Proof:

$$\begin{array}{ll} & s \rightarrow t \\ \Leftrightarrow & \left\{ \begin{array}{l} \text{by defn of a chain, and since} \rightarrow \text{is the} \\ \text{transitive closure of} \rightsquigarrow \cup \prec_{im} \end{array} \right\} \\ & (\exists c :: first(c) = s \ \wedge \ last(c) = t) \\ \Leftrightarrow & \{ \text{defn of } ml \} \\ & (\exists k : k > 0 : ml(s,t) = k) \\ \Leftrightarrow & \{ \text{defn of } \xrightarrow{k} \} \\ & (\exists k : k > 0 : s \xrightarrow{k} t) \end{array} \quad \square$$

Lemma 8 $s \xrightarrow{1} t \Rightarrow s \prec_{im} t \ \vee \ s \rightsquigarrow t$

Proof:

$$\begin{array}{ll} & s \xrightarrow{1} t \\ \Rightarrow & \{ \text{defn of } \xrightarrow{k} \} \\ & ml(s,t) = 1 \\ \Rightarrow & \{ \text{defn of } ml \} \\ & (\exists c :: first(c) = s \ \wedge \ last(c) = t \ \wedge \ len(c) = 1) \\ \Rightarrow & \{ \text{defn of a chain} \} \\ & s \prec_{im} t \ \vee \ s \rightsquigarrow t \end{array}$$

$\square$

The converse of lemma 8 does not hold in the case where messages are received out of order. For example, in figure 4 $s1 \rightsquigarrow t4$ holds but not $s1 \xrightarrow{1} t4$ does not. The reason is that there is a chain of length four from $s1$ to $t4$, thus $s1 \xrightarrow{4} t4$.

Lemma 9 $s \xrightarrow{k} t \ \wedge \ (k > 1) \Rightarrow (\exists u :: s \xrightarrow{k-1} u \ \wedge \ u \xrightarrow{1} t)$

Proof:

$$\begin{array}{ll}
 & s \xrightarrow{k} t \ \wedge\ (k > 1) \\
\Rightarrow & \{\text{ defn of } \xrightarrow{k} \} \\
 & ml(s,t) = k \ \wedge\ k > 1 \\
\Rightarrow & \{\text{ chain lemma }\} \\
 & (\exists u :: ml(s,u) = k-1 \ \wedge\ ml(u,t) = 1) \\
\Rightarrow & \{\text{ defn of } \xrightarrow{k} \} \\
 & (\exists u :: s \xrightarrow{k-1} u \ \wedge\ u \xrightarrow{1} t)
\end{array}$$

□

The proof technique for $\rightarrow$ outlined above is fairly intuitive since $\rightarrow$ is defined using transitive closure. The inductive proof technique for $\not\rightarrow$ is more interesting. We define for $k \geq 0$:

$$s \overset{k}{\not\rightarrow} t \triangleq s \not\rightarrow t \ \wedge\ ml(Init, t) = k$$

Thus $s \overset{k}{\not\rightarrow} t$ if and only if $s \not\rightarrow t$ and the longest chain from some initial state to t has length k.

Lemmas 10, 11 and 12 are used in inductive proofs for properties stated with the $\overset{k}{\not\rightarrow}$ relation. The method is similar to the one described above for $\xrightarrow{k}$. Suppose $s \not\rightarrow t \Rightarrow R(s,t)$. Lemma 10 tells us that it is sufficient to prove $s \overset{k}{\not\rightarrow} t \Rightarrow R(s,t)$ for all $k \geq 0$, which can be proven by induction on k. To prove the base case, $s \overset{0}{\not\rightarrow} t$, we need to show that $R(s,t)$ holds when t is an initial state. The inductive case ($k > 0$) uses lemma 12. It is not immediately obvious that this lemma applies in the inductive case, but consider the following. The assumption in the inductive case is $k > 0 \ \wedge\ s \overset{k}{\not\rightarrow} t$. This implies that t is not an initial state (see lemma 11), which in turn implies that there exists some state u such that $u \rightarrow t$. Thus the left hand side of lemma 12 is true in the induction case.

Lemma 10 $s \not\rightarrow t \Leftrightarrow (\exists k : k \geq 0 : s \overset{k}{\not\rightarrow} t)$

Proof:

$$\begin{array}{ll}
 & s \not\rightarrow t \\
\Leftrightarrow & \{\text{ by defn of } ml(Init, t) \} \\
 & s \not\rightarrow t \ \wedge\ ml(Init, t) \geq 0 \\
\Leftrightarrow & \{\text{ defn of } \xrightarrow{k} \} \\
 & (\exists k : k \geq 0 : s \overset{k}{\not\rightarrow} t)
\end{array}$$

□

Lemma 11 $s \overset{0}{\not\rightarrow} t \Leftrightarrow Init(t)$

Proof:

$$s \overset{0}{\not\rightarrow} t$$

$\Leftrightarrow$ { defn of $\overset{0}{\not\rightarrow}$ }

$ml(Init, t) = 0 \ \wedge \ s \not\rightarrow t$

$\Leftrightarrow$ { defn of $ml(Init, t)$ }

$\neg(\exists u :: u \rightarrow t) \ \wedge \ s \not\rightarrow t$

$\Leftrightarrow$ { left conjunct implies right conjunct }

$\neg(\exists u :: u \rightarrow t)$

$\Leftrightarrow$ { defn of $Init(t)$ }

$Init(t)$

$\square$

Lemma 12 $k > 0 \ \wedge \ s \overset{k}{\not\rightarrow} t \ \wedge \ u \rightarrow t \Rightarrow (\exists j : 0 \leq j < k : s \overset{j}{\not\rightarrow} u)$

Proof:

$k > 0 \ \wedge \ s \overset{k}{\not\rightarrow} t \ \wedge \ u \rightarrow t$

$\Rightarrow$ { otherwise $s \rightarrow t$ }

$k > 0 \ \wedge \ s \not\rightarrow u \ \wedge \ s \overset{k}{\not\rightarrow} t$

$\Rightarrow$ { defn of $\overset{k}{\not\rightarrow}$ }

$k > 0 \ \wedge \ s \not\rightarrow u \ \wedge \ ml(Init, t) = k$

$\Rightarrow$ { otherwise $ml(Init, u) > k$ }

$k > 0 \ \wedge \ s \not\rightarrow u \ \wedge \ ml(Init, u) < k$

$\Rightarrow$ { defn of $\overset{j}{\not\rightarrow}$ }

$(\exists j : 0 \leq j < k : s \overset{j}{\not\rightarrow} u)$

$\square$

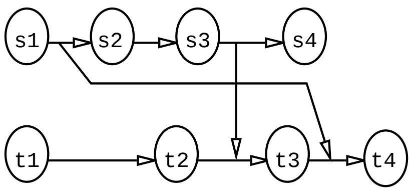


Figure 4: Some examples of our relations on states: $s3 \overset{1}{\rightarrow} t3$, $t1 \overset{2}{\rightarrow} t3$, $s1 \overset{4}{\rightarrow} t4$, $s2 \overset{0}{\not\rightarrow} t1$, $s4 \overset{3}{\not\rightarrow} t3$.

The traditional vector clock algorithm assigns a vector $s.v$ to every local state s such that $s.v < t.v$ if and only if $s \rightarrow t$. We use a slightly different version in which this condition holds when s and t are on different processes. We use this version because it is practical: It conserves state space since the vector components are incremented less frequently; and in general, one is interested in causal relationships between states on different processes. The version we use maintains the following property:

$$(\forall s, t : s.p \neq t.p : s.v < t.v \Leftrightarrow s \rightarrow t)$$

The algorithm is described by the initial conditions and the actions taken for each event type. We have also shown within the braces, the property of the event that is used in proving correctness of the algorithm.

For any initial state s:
  for $i := 1$ to N
    $s.v[i] := 0$;
  $s.v[s.p] := 1$;
  $\{(\forall i : i \neq s.p : s.v[i] = 0) \ \wedge \ (s.v[s.p] = 1) \ \}$

Rule for a send event (s, snd, t):
  $t.v := s.v$;
  $t.v[t.p] + +$;
  $\{(\forall i : i \neq t.p : t.v[i] = s.v[i]) \ \wedge \ t.v[t.p] > s.v[t.p])\}$

Rule for a receive event $(s, rcv(u), t)$:
  for $i := 1$ to N
    $t.v[i] := \max(s.v[i], u.v[i])$;
  $\{(\forall i :: t.v[i] = \max(s.v[i], u.v[i])) \ \}$

Rule for an internal event (s, int, t):
  $t.v := s.v$;
  $\{t.v = s.v\}$

The version presented above is harder to prove than the traditional algorithm because of the message receive action. In the traditional algorithm, when a message is received in state s, the local clock, $s.p$, is incremented. This ensures that $(s, rcv(u), t)$ implies $s.v < t.v$ and $u.v < t.v$. The action taken in this version, $t.v := max(s.v, u.v)$, does not imply $s.v < t.v$ nor does it imply $u.v < t.v$. This makes this version significantly more difficult to prove.

In our proof, we use properties of the algorithm given as comments in the text. Their validity is clear from the algorithm text. Thus, the proof is valid for any algorithm which satisfies these properties. For example, in the send rule of the algorithm, $t.v[t.p]$ could be increased by any positive amount and our proof would still be valid.

4.3 Proof of Vector Clock Algorithm

In this section we prove the property stated earlier: $(\forall s, t : s.p \neq t.p : s.v < t.v \Leftrightarrow s \rightarrow t)$. This is accomplished by proving the following claims:

$$s.p \neq t.p \ \wedge \ s \rightarrow t \Rightarrow s.v < t.v \tag{1}$$

$$s.p \neq t.p \ \wedge \ s.v < t.v \Rightarrow s \rightarrow t \tag{2}$$

Lemma 13 states that if there is a chain of events from s to t then $s.v \leq t.v$. In the traditional algorithm, proof of the property $s \rightarrow t \Rightarrow s.v < t.v$ (which does not hold here) is essentially the same as this proof. This is because, in the traditional algorithm, local clocks are incremented for every event type. Note also that the proof of lemma 13 does not use the initial conditions. Thus the lemma holds independent of the initial values of the vectors.

Lemma 13 $s \rightarrow t \Rightarrow s.v \leq t.v$

Proof: It is sufficient to show that for all $k > 0$: $s \xrightarrow{k} t \Rightarrow s.v \leq t.v$. We use induction on k.

Base $(k = 1)$:

$$
\begin{array}{ll}
 & s \xrightarrow{1} t \\
\Rightarrow & \{ \text{ lemma 8 } \} \\
 & s \prec_{im} t \;\vee\; s \leadsto t \\
\Rightarrow & \{ \text{ expand } s \prec_{im} t \text{ and } s \leadsto t \;\} \\
 & (s, int, t) \;\vee\; (s, snd, t) \;\vee\; (\exists u :: (s, rcv(u), t)) \\
 & \;\vee\; (\exists u :: (u, rcv(s), t)) \\
\Rightarrow & \{ \text{ Snd, Rcv, and Int rules } \} \\
 & (s.v = t.v) \;\vee\; (s.v < t.v) \;\vee\; (s.v \leq t.v) \\
 & \;\vee\; (s.v \leq t.v) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & s.v \leq t.v
\end{array}
$$

Induction: $(k > 1)$

$$
\begin{array}{ll}
 & s \xrightarrow{k} t \;\wedge\; (k > 1) \\
\Rightarrow & \{ \text{ lemma 9 } \} \\
 & (\exists u :: s \xrightarrow{k-1} u \;\wedge\; u \xrightarrow{1} t) \\
\Rightarrow & \{ \text{ induction hypothesis } \} \\
 & (\exists u :: s.v \leq u.v \;\wedge\; u.v \leq t.v) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & s.v \leq t.v
\end{array}
$$

□

Lemma 14 states that if two states s and t are on different processes, and s does not causally precede t, then $t.v[s.p] < s.v[s.p]$. Our formal proof of this lemma is nontrivial. This proof is by induction on k in the $\stackrel{k}{\not\rightarrow}$ relation.

Lemma 14 $(\forall s, t : s.p \neq t.p : s \not\rightarrow t \Rightarrow t.v[s.p] < s.v[s.p])$

Proof: It is sufficient to show that for all $k \geq 0$: $s \stackrel{k}{\not\rightarrow} t \;\wedge\; s.p \neq t.p \Rightarrow t.v[s.p] < s.v[s.p]$. We use induction on k.

Base $(k = 0)$:

$$
\begin{array}{ll}
 & s \overset{0}{\not\to} t \ \wedge\ s.p \neq t.p \\
\Rightarrow & \{ \text{ lemma 11 } \} \\
 & Init(t) \ \wedge\ s.p \neq t.p \\
\Rightarrow & \{ \text{ let } u \text{ be initial state in } s.p \ \} \\
 & Init(t) \ \wedge\ s.p \neq t.p \ \wedge \\
 & (\exists u : Init(u) \ \wedge\ u.p = s.p : u = s \ \vee\ u \to s) \\
\Rightarrow & \{ \text{ lemma 13 } \} \\
 & Init(t) \ \wedge\ s.p \neq t.p \ \wedge \\
 & (\exists u : Init(u) \ \wedge\ u.p = s.p : u.v = s.v \ \vee\ u.v \leq s.v) \\
\Rightarrow & \{ \text{ Init rule } \} \\
 & t.v[s.p] = 0 \\
 & \ \wedge\ (\exists u : u.v[s.p] = 1 : u.v = s.v \ \vee\ u.v \leq s.v) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p]
\end{array}
$$

Induction: $(k > 0)$

$$
\begin{array}{ll}
 & s \overset{k}{\not\to} t \ \wedge\ s.p \neq t.p \ \wedge\ k > 0 \\
\Rightarrow & \{ \text{ let } u \text{ satisfy } u \prec_{im} t,\ u \text{ exists since } \neg Init(t) \ \} \\
 & s \overset{k}{\not\to} t \ \wedge\ s.p \neq t.p \ \wedge\ u.p = t.p \ \wedge\ u \prec_{im} t \\
\Rightarrow & \{ \text{ lemma 12 } \} \\
 & s \overset{j}{\not\to} u \ \wedge\ 0 \leq j < k \ \wedge\ u.p \neq s.p \ \wedge\ u \prec_{im} t \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
 & u.v[s.p] < s.v[s.p] \ \wedge\ u \prec_{im} t \\
\Rightarrow & \{ \text{ expand } u \prec_{im} t \ \} \\
 & u.v[s.p] < s.v[s.p] \\
 & \ \wedge\ ((u, int, t) \ \vee\ (u, snd, t) \ \vee\ (u, rcv(w), t))
\end{array}
$$

Consider each disjunct separately:

Case 1: (u, int, t)

$$
\begin{array}{ll}
 & u.v[s.p] < s.v[s.p] \ \wedge\ (u, int, t) \\
\Rightarrow & \{ \text{ Int rule } \} \\
 & u.v[s.p] < s.v[s.p] \ \wedge\ t.v = u.v \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p]
\end{array}
$$

Case 2: (u, snd, t)

$$
\begin{array}{ll}
 & u.v[s.p] < s.v[s.p] \ \wedge\ (u, snd, t) \\
\Rightarrow & \{ \text{ Snd rule, } s.p \neq t.p \ \} \\
 & u.v[s.p] < s.v[s.p] \ \wedge\ t.v[s.p] = u.v[s.p] \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p]
\end{array}
$$

Case 3: $(u, rcv(w), t)$

$$
\begin{array}{ll}
& u.v[s.p] < s.v[s.p] \;\wedge\; (u, rcv(w), t) \\
\Rightarrow & \{ \text{ Rcv rule } \} \\
& u.v[s.p] < s.v[s.p] \;\wedge\; (u, rcv(w), t) \\
& \;\wedge\; (t.v[s.p] = u.v[s.p] \;\vee\; t.v[s.p] = w.v[s.p]) \\
\Rightarrow & \{ \text{ simplify } \} \\
& t.v[s.p] < s.v[s.p] \\
& \;\vee\; ((u, rcv(w), t) \;\wedge\; t.v[s.p] = w.v[s.p])
\end{array}
$$

For case 3, it suffices to prove the following two cases.

Case 3A: $w.p = s.p$

$$
\begin{array}{ll}
& t.v[s.p] = w.v[s.p] \;\wedge\; (u, rcv(w), t) \;\wedge\; w.p = s.p \\
\Rightarrow & \left\{ \begin{array}{l} \text{let } x \text{ satisfy } w \prec_{im} x,\ x \text{ exists since} \\ w \rightsquigarrow t \text{ implies } \neg Final(w) \end{array} \right\} \\
& t.v[s.p] = w.v[s.p] \;\wedge\; (w, snd, x) \;\wedge\; w.p = s.p \\
\Rightarrow & \{ \text{ otherwise } s \rightarrow t \ \} \\
& t.v[s.p] = w.v[s.p] \;\wedge\; (w, snd, x) \;\wedge\; w.p = s.p \\
& \;\wedge\; w \rightarrow s \\
\Rightarrow & \{ \text{ since } w \prec_{im} x \ \} \\
& t.v[s.p] = w.v[s.p] \;\wedge\; (w, snd, x) \;\wedge\; w.p = s.p \\
& \;\wedge\; (x = s \;\vee\; x \rightarrow s) \\
\Rightarrow & \{ \text{ Snd rule } \} \\
& t.v[s.p] = w.v[s.p] \;\wedge\; w.v[s.p] < x.v[s.p] \\
& \;\wedge\; (x = s \;\vee\; x \rightarrow s) \\
\Rightarrow & \{ \text{ lemma 13 } \} \\
& t.v[s.p] = w.v[s.p] \;\wedge\; w.v[s.p] < x.v[s.p] \\
& \;\wedge\; x.v \leq s.v \\
\Rightarrow & \{ \text{ simplify } \} \\
& t.v[s.p] < s.v[s.p]
\end{array}
$$

Case 3B: $w.p \neq s.p$

$$
\begin{array}{ll}
& t.v[s.p] = w.v[s.p] \;\wedge\; (u, rcv(w), t) \;\wedge\; w.p \neq s.p \\
\Rightarrow & \{ \text{ use } s \stackrel{k}{\not\rightarrow} t,\ k > 0, \text{ and lemma 12 } \} \\
& t.v[s.p] = w.v[s.p] \;\wedge\; w.p \neq s.p \;\wedge\; s \stackrel{j}{\not\rightarrow} w \\
& \;\wedge\; 0 \leq j < k \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
& t.v[s.p] = w.v[s.p] \;\wedge\; w.v[s.p] < s.v[s.p] \\
\Rightarrow & \{ \text{ simplify } \} \\
& t.v[s.p] < s.v[s.p]
\end{array}
$$

□

Lemma 15 is a refinement of lemma 13 for the case when $s.p \neq t.p$, in which case $s.v < t.v$.

Lemma 15 $(\forall s, t : s.p \neq t.p : s \rightarrow t \Rightarrow s.v < t.v)$

Proof: It is sufficient to show that for all $k > 0$: $s \xrightarrow{k} t \;\wedge\; s.p \neq t.p \Rightarrow t.v < s.v$. We use induction on k.

Base $(k = 1)$:

$$
\begin{array}{ll}
 & s \xrightarrow{1} t \ \wedge \ s.p \neq t.p \\
\Rightarrow & \{ \text{ defn of } \xrightarrow{1} \text{ and lemma 8 } \} \\
 & s \leadsto t \ \wedge \ s.p \neq t.p \\
\Rightarrow & \{ \text{ let } u \text{ satisfy } u \prec_{im} t \ \} \\
 & s.p \neq u.p \ \wedge \ (u, rcv(s), t) \\
\Rightarrow & \left\{ \begin{array}{l} \text{otherwise } t \rightarrow s \text{ (since there is only one} \\ \text{event between } u \text{ and } t) \end{array} \right\} \\
 & u \not\rightarrow s \ \wedge \ s.p \neq u.p \ \wedge \ (u, rcv(s), t) \\
\Rightarrow & \{ \text{ lemma 14 and rcv rule } \} \\
 & s.v[s.p] < u.v[s.p] \\
 & \ \wedge \ (\forall i :: t.v[i] = \max(u.v[i], s.v[i])) \\
\Rightarrow & \\
 & s.v < t.v
\end{array}
$$

Induction $(k > 0)$:

$$
\begin{array}{ll}
 & s \xrightarrow{k} t \ \wedge \ k > 0 \ \wedge \ s.p \neq t.p \\
\Rightarrow & \{ \text{ lemma 9 } \} \\
 & (\exists u :: s \xrightarrow{k-1} u \ \wedge \ u \xrightarrow{1} t \ \wedge \ s.p \neq t.p) \\
\Rightarrow & \{ \ u.p \text{ cannot have two values } \} \\
 & (\exists u :: s \xrightarrow{k-1} u \ \wedge \ u \xrightarrow{1} t \ \wedge \ (u.p \neq t.p \ \vee \ u.p \neq s.p)) \\
\Rightarrow & \\
 & (\exists u :: (s \xrightarrow{k-1} u \ \wedge \ u \xrightarrow{1} t \ \wedge \ u.p \neq t.p) \ \vee \ (s \xrightarrow{k-1} u \ \wedge \ u \xrightarrow{1} t \ \wedge \ u.p \neq s.p)) \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
 & (\exists u :: (s \xrightarrow{k-1} u \ \wedge \ u.v < t.v) \vee \ (s.v < u.v \ \wedge \ u \xrightarrow{1} t)) \\
\Rightarrow & \{ \text{ lemma 13 } \} \\
 & (\exists u :: (s.v \leq u.v \ \wedge \ u.v < t.v) \vee \ (s.v < u.v \ \wedge \ u.v \leq t.v)) \\
\Rightarrow & \\
 & s.v < t.v
\end{array}
$$

□

Theorem 16 states the property which we set out to prove at the beginning of this section.

Theorem 16 $(\forall s, t : s.p \neq t.p : s \rightarrow t \Leftrightarrow s.v < t.v)$

Proof: Immediate from Lemmas 14 and 15. □

5 Direct Dependency Clocks

5.1 Algorithm

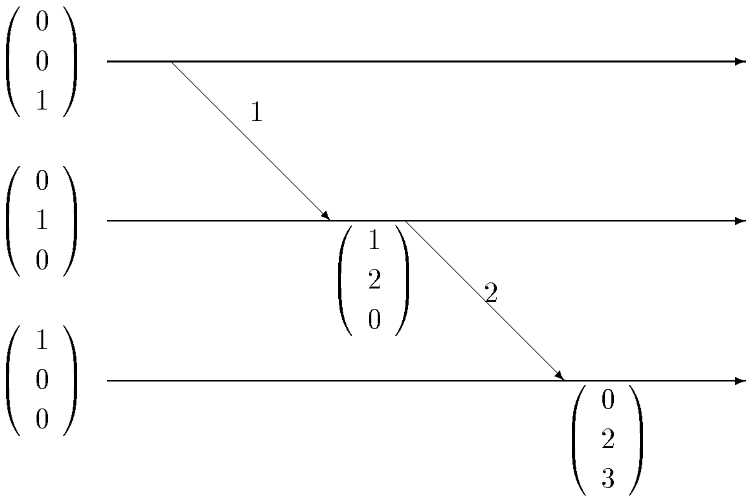
One drawback with the vector clock algorithm is that it requires $O(N)$ integers with every message. For many applications, a weaker version of the clock suffices. We now describe a clock algorithm that is used by many algorithms in distributed systems. These clocks require only one integer to be appended to each message. We will call these clocks direct-dependency clocks. The algorithm is described by the initial conditions and the actions taken for each event type.

For any initial state s:
$(\forall i : i \neq s.p : s.v[i] = 0) \ \wedge \ (s.v[s.p] = 1)$

Rule for a send event (s, snd, t) or an internal (s, int, t):
$t.v[t.p] := s.v[t.p] + 1;$

Rule for a receive event $(s, rcv(u), t)$:
$t.v[t.p] := \max(s.v[t.p], u.v[u.p]) + 1;$
$t.v[u.p] := \max(u.v[u.p], s.v[u.p]);$

An example of a distributed computation and its associated direct dependency clock is given below.



5.2 Properties of Direct Dependency Clocks

We first observe that if we retain only the ith component for the ith process, then the above algorithm is identical to logical clock algorithm. We provide a direct proof of the logical clock property of the direct dependency clocks.

Lemma 17 $s \rightarrow t \Rightarrow s.v[s.p] < t.v[t.p]$

Proof: It is sufficient to show that for all $k > 0$: $s \xrightarrow{k} t \Rightarrow s.v[s.p] < t.v[t.p]$. We use induction on k.

Base $(k = 1)$:

$$
\begin{array}{ll}
& s \xrightarrow{1} t \\
\Rightarrow & \{ \text{ lemma Base-case } \rightarrow \} \\
& s \prec_{im} t \ \vee \ s \rightsquigarrow t \\
\Rightarrow & \{ \text{ expand } s \prec_{im} t \text{ and } s \rightsquigarrow t \ \} \\
& (s, int, t) \ \vee \ (s, snd, t) \ \vee \ (\exists u :: (s, rcv(u), t)) \\
& \ \vee \ (\exists u :: (u, rcv(s), t)) \\
\Rightarrow & \{ \text{ Snd, Rcv, and Int rules } \} \\
& s.v[s.p] < t.v[t.p]
\end{array}
$$

Induction: $(k > 1)$

$$
\begin{array}{ll}
& s \xrightarrow{k} t \ \wedge \ (k > 1) \\
\Rightarrow & \{ \text{ lemma induction } \rightarrow \} \\
& (\exists u :: s \xrightarrow{k-1} u \ \wedge \ u \xrightarrow{1} t) \\
\Rightarrow & \{ \text{ induction hypothesis } \} \\
& (\exists u :: s.v[s.p] < u.v[u.p] \ \wedge \ u.v[u.p] < t.v[t.p]) \\
\Rightarrow & \{ \text{ simplify } \} \\
& s.v[s.p] < t.v[t.p]
\end{array}
$$

$\square$

However, our interest in direct dependency clocks is not due to its logical clock property (Lamport's logical clock is sufficient for that), but for allowing it to capture the notion of direct dependency. We first define a relation $\rightarrow_d$, a subset of $\rightarrow$, as follows:

$$s \rightarrow_d t \equiv s \prec t \ \vee \ \exists q, r : s \preceq q \ \wedge \ q \rightsquigarrow r \ \wedge \ r \preceq t$$

The following property makes direct dependency clocks useful for many applications.

Lemma 18 *$\forall s, t : s.p \neq t.p : (s \rightarrow_d t) \Leftrightarrow (s.v[s.p] \leq t.v[s.p])$.*

Proof: To prove this lemma formally in the forward direction, we use induction on the length of the chain between s and t as we did for $\rightarrow$. However, for $\rightarrow_d$ we will restrict our attention to only those chains that go only through processes $s.p$ and $t.p$. It is easy to see that each event (q,e,r) in this chain satisfies $q.v[q.p] \leq r.v[q.p]$. We leave the details to the reader.

To prove the converse formally, we use induction on the length of the rank of t where rank is defined as:

$$rank(t) = |\{q | q \prec t\}|$$

That is, the rank of a local state t is the number of states that locally precede t. Let k be the rank of t. Base $(k = 0)$:

$$
\begin{array}{ll}
 & rank(t) = 0 \\
\Rightarrow & \{ \text{ definition of rank } \} \\
 & Init(t) \\
\Rightarrow & \{ \text{ let } u \text{ be initial state in } s.p \ \} \\
 & Init(t) \ \wedge \\
 & (\exists u : Init(u) \ \wedge \ u.p = s.p : u = s \ \vee \ u \rightarrow s) \\
\Rightarrow & \{ \text{ lemma 17 } \} \\
 & Init(t) \ \wedge \\
 & (\exists u : Init(u) \ \wedge \ u.p = s.p : u.v = s.v \ \vee \ u.v[u.p] < s.v[u.p]) \\
\Rightarrow & \{ \text{ Init rule } \} \\
 & t.v[s.p] = 0 \\
 & \ \wedge \ (\exists u : u.v[s.p] = 1 : u.v[s.p] \leq s.v[s.p]) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & \neg(s.v[s.p] \leq t.v[s.p])
\end{array}
$$

Induction: $(k > 0)$

$$
\begin{array}{ll}
 & s \not\rightarrow_d t \ \wedge \ k > 0 \\
\Rightarrow & \{ \text{ let } u \text{ satisfy } u \prec_{im} t, \ u \text{ exists since } k = rank(t) > 0 \ \} \\
 & s \not\rightarrow_d t \ \wedge \ u \prec_{im} t \\
\Rightarrow & \{ \ s \rightarrow_d u \text{ implies } s \rightarrow_d t \ \} \\
 & s \not\rightarrow_d u \ \wedge \ rank(u) < k \ \wedge \ u \prec_{im} t \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
 & \neg(s.v[s.p] \leq u.v[s.p]) \ \wedge \ u \prec_{im} t \\
\Rightarrow & \{ \text{ expand } u \prec_{im} t \ \} \\
 & \neg(s.v[s.p] \leq u.v[s.p]) \\
 & \ \wedge \ ((u, int, t) \ \vee \ (u, snd, t) \ \vee \ (u, rcv(w), t))
\end{array}
$$

Consider each disjunct separately:

Case 1: $(u, int, t) \ \vee \ (u, snd, t)$

$$
\begin{array}{ll}
 & (s.v[s.p] > u.v[s.p]) \ \wedge \ ((u, int, t) \ \vee \ (u, snd, t)) \\
\Rightarrow & \{ \text{ Int rule, Snd Rule } \} \\
 & (s.v[s.p] > u.v[s.p]) \ \wedge \ t.v[s.p] = u.v[s.p] \\
\Rightarrow & \{ \text{ simplify } \} \\
 & (s.v[s.p] > t.v[s.p])
\end{array}
$$

Case 2: $(u, rcv(w), t), w.p = s.p$

$$
\begin{array}{ll}
 & s \not\rightarrow_d t, s.p = w.p, w \rightsquigarrow t \\
\Rightarrow & \{ \; s \prec w \text{ or } s = w \text{ imply } s \rightarrow_d t \; \} \\
 & w \prec s \\
\Rightarrow & \{ \text{ from program, local component incremented after send } \} \\
 & w.v[s.p] < s.v[s.p] \\
\Rightarrow & \{ \text{ Rcv rule } \} \\
 & (s.v[s.p] > w.v[s.p]) \;\wedge\; ((w.v[s.p] = t.v[s.p]) \;\vee\; (u.v[s.p] = t.v[s.p])) \\
\Rightarrow & \{ \; (s.v[s.p] > u.v[s.p]) \; \} \\
 & (s.v[s.p] > t.v[s.p])
\end{array}
$$

Case 3: $(u, rcv(w), t), w.p \neq s.p$

$$
\begin{array}{ll}
 & (s.v[s.p] > u.v[s.p]) \;\wedge\; (u, rcv(w), t) \;\wedge\; w.p \neq s.p \\
\Rightarrow & \{RecvRule\} \\
 & (s.v[s.p] > u.v[s.p]) \;\wedge\; t.v[s.p] = u.v[s.p] \\
\Rightarrow & \{ \text{ simplify } \} \\
 & (s.v[s.p] > t.v[s.p])
\end{array}
$$

□

The following is an easy corollary.

Corollary 19 $(\forall s, t : s.p \neq t.p : s \not\rightarrow t \Rightarrow \neg(s.v[s.p] > t.v[s.p])$

Proof: Observe that $s \not\rightarrow t$ implies $s \not\rightarrow_d t$. □

6 Higher Dimensional Clocks

It is natural to ask whether using higher dimensional clocks can give processes additional knowledge. The answer is - yes. Assume that "knowledge" is a collection of facts. By proper encoding, any finite amount of knowledge can be represented by an integer. So, now we assume that the knowledge function K returns the knowledge the process has about itself as a natural number, i.e. $K(s) \in \mathcal{N}$ for any s.

We assume that a process never forgets, i.e. the knowledge increases with time for any process. Also, the only way knowledge can be communicated to a different process is through a message. If we ensure that any process includes all it knows in the message, and on receiving any message from P_j, the process P_i updates its own knowledge. Then, its knowledge is greater than the knowledge sent by P_j and its own previous knowledge. The above can be concisely stated as "knowledge is monotonic with respect to $\rightarrow$." Formally, $s \rightarrow t \Rightarrow K(s) < K(t)$. The alert reader would notice the similarity between these rules and those of logical clocks. The only difference is that the $<$ may not be the natural less than relation on the numbers.

Now let us say that for some reason, it is important for a process to know not only what it knows but also what other people know. A time-vector can therefore be alternatively viewed as a knowledge vector. In this interpretation, $s.v[i]$ denotes what process $s.p$ knows about the process i in the local state s. If processes include their entire vector in their messages, we retain monotonicity with respect to $\rightarrow$.

In some applications it may be important for the process to know a still higher level of knowledge. Now $s.v[i,j]$ would represent what process $s.p$ knows about what process i knows about process j. For example, if $s.v[i, s.p] > k$ for all i, then process $s.p$ can conclude that everybody knows that its state is strictly greater than k. This may be important to discard some information which has now been broadcast to everyone (in spite of uncertain communication).

Next, we describe a matrix clock. The discussion of properties of a matrix clock requires the notion of intervals, and predecessor and the successor function.

6.1 State Interval

An external event is the sending or receiving of a message. The n^{th} interval in P_i (denoted by (i,n)) is the subchain of $(S_i, \rightarrow)$ between the $(n-1)^{th}$ and n^{th} external events. For a given interval (i,n), if n is out of range then (i,n) refers to $\perp$ or $\top$. The notion of intervals is useful because the relation of two states belonging to the same interval is a congruence with respect to $\rightarrow$. Thus, for any two states s, s' in the same interval and any state u: $(s \rightarrow u \iff s' \rightarrow u)$ and $(u \rightarrow s \iff u \rightarrow s')$. We exploit this congruence in our algorithms by assigning a single timestamp to all states belonging to the same interval. Note that the notion of state cuts can be extended to *interval cuts*. An example of a distributed computation and its state intervals is shown in Figure 5.

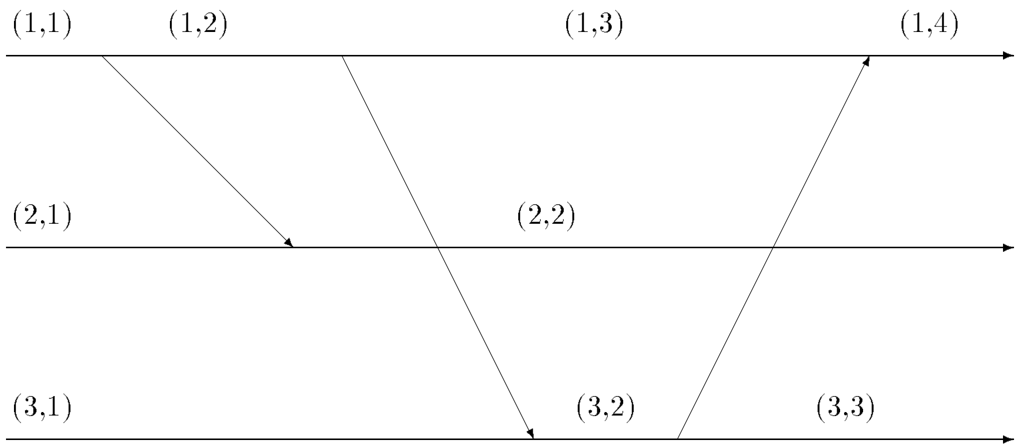


Figure 5: State intervals

6.2 Predecessor and Successor functions

The predecessor and successor functions are defined as follows for $u \in S$ and $0 \leq i \leq N$:

$$pred.u.i = (\max v : v \in S_i \ \wedge \ v \rightarrow u : v)$$

$$succ.u.i = (\min v : v \in S_i \ \wedge \ u \rightarrow v : v)$$

Thus if $(pred.u.i = v)$ then v is the maximum element in $(S_i, \rightarrow)$ which happened before u, or $\perp$ if no element in S_i happened before u.

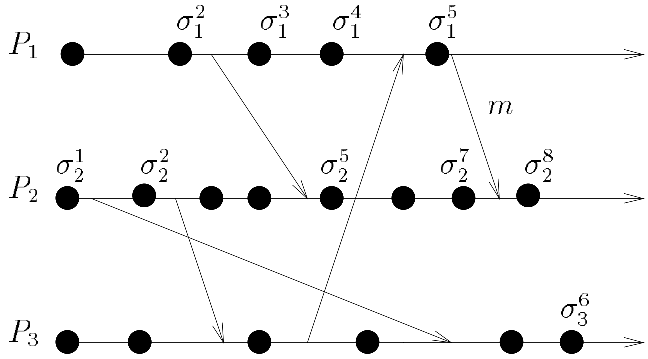


Figure 6: Demonstration of the *pred* function: $pred.\sigma_1^5.2 = \sigma_2^2$, $pred.\sigma_3^6.2 = \sigma_2^2$, $pred.\sigma_1^4.2 = \perp$.

Due to the congruence for states belonging to the same interval, the *pred* and *succ* functions and the $||$ relation are well defined on intervals.

Consider a state $\sigma_i \in S_i$. The predecessor of σ_i in P_j, denoted $pred.\sigma_i.j$, is the latest state in P_j which happens before σ_i. Figure 6 shows an example. Notice that the predecessor or σ_i^n on P_i is just the previous state σ_i^{n-1}.

6.3 Algorithm for Matrix Clock

For any initial state s:

$$(\forall i, j : i \neq s.p \vee j \neq s.p : s.v[i,j] = 0) \ \wedge \ (s.v[s.p, s.p] = 1)$$

Rule for a send event (s, snd, t) :

$$t.v[t.p, t.p] := s.v[s.p, s.p] + 1;$$

Rule for a receive event $(s, rcv(u), t)$:

$$t.v := \max(s.v, u.v);$$
$$t.v[t.p, .] := \max(s.v[s.p, .], u.v[u.p, .]);$$
$$t.v[t.p, t.p] + +;$$

Figure 7: Matrix Clock Algorithm

The matrix clock algorithm is presented in figure 7. The algorithm is easier to understand by noticing the vector clock algorithm embedded within it. If the row index is held constant, then it reduces to the vector clock algorithm. Figure 8 shows values of the matrix clock and message tags on an example run. Note that row 1 of P_1's matrix is a traditional vector clock restricted to indices 1 and 2, and row 2 equals the value of P_2's vector clock at a state in the "past" of P_1. Similar properties hold for P_2's matrix clock.

Let M_k^n denote the value of the matrix clock in interval (k, n). The following description applies to an $N \times N$ matrix clock in a system with N processes. We describe the information contained

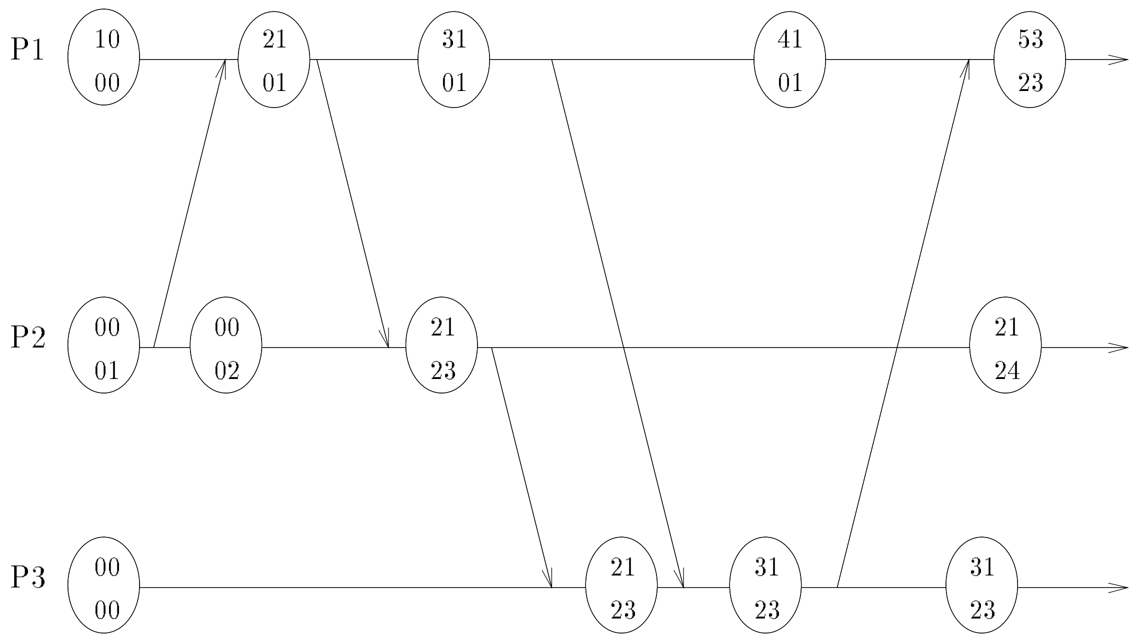


Figure 8: Matrix clock example.

in a matrix clock M_k^n at interval (k, n) in three steps. First we explain the meaning of the k^{th} diagonal element. Then we explain the meaning of the k^{th} row, followed by the entire matrix.

The k^{th} diagonal element, $M_k^n[k,k]$, is an interval counter at P_k. Thus for any interval (k, n), the following is true: $M_k^n[k,k] = n$. Row k of M_k^n is equivalent to a traditional vector clock. In fact, when we say "P_k's vector clock" we are referring to row k of P_k's matrix clock. Since the vector clock implements the *pred* function, row k of M_k^n can be used to find predecessors of $(k, n) = (k, M_k^n[k,k])$ as follows: the predecessor of $(k, M_k^n[k,k])$ on process j is the interval $(j, M_k^n[k,j])$. In fact, this applies to all rows: the predecessor of $(i, M_k^n[i,i])$ on process j is the interval $(j, M_k^n[i,j])$. Furthermore, row k of M_k^n equals the diagonal of M_k^n. Thus we can use row k to find the $pred.(k,n).j$ for $j \neq k$, and then use row j to find $pred.(pred.(k,n).j).i$ for $i \neq j$.

The matrix clock can be summarized in three simple rules. First, the k^{th} diagonal element corresponds to some interval on P_k. Second, row i is the value of P_i's vector clock in interval $(i, M_k[i,i])$ and third, row k of M_k equals the diagonal. The meaning of a matrix clock is formally stated and proven in lemma 20.

Lemma 20 *(M1)* $M_k^n[k,k] = n$
(M2) $i \neq j \Rightarrow (j, M_k^n[i,j]) = pred.(i, M_k^n[i,i]).j$
(M3) $M_k^n[i,i] = M_k^n[k,i]$

Proof: Each part is proven by induction. In the base case, $n = 1$ and (k, n) is an initial interval. Induction applies to $n > 1$.

(M1) **Base** ($n = 1$): Initial value of $M_k^n[k,k] = 1$.
Induction ($n > 1$): We assume $M_k^n[k,k] = n$ and show that $M_k^{n+1}[k,k] = n+1$. P_k enters

interval $(k, n+1)$ only after sending or receiving a message. From the program text it is clear the k^{th} diagonal element is incremented by one. Thus $M_k^{n+1}[k,k] = M_k^n[k,k] + 1 = n+1$.

(M2) **Base** ($n = 1$): Since $n = 1$, M_k^n is the initial value of the matrix clock in P_k. Thus, $i \neq j$ implies that $M_k^n[i,j] = 0$, and hence $(j, M_k^n[i,j]) = (j, 0) = \perp$. Thus we need to show $pred.(i, M_k^n[i,i]).j = \perp$ also. Suppose $i \neq k$, then by initial value of M_k^n, we know $(i, M_k^n[i,i]) = (i, 0)$, thus $pred.(i, M_k^n[i,i]).j = pred.(i,0).j = \perp$. Now suppose $i = k$, then by initial value of $M_k^n[i,i]$ we know that $(i, M_k^n[i,i]) = (i, 1)$. Since no interval precedes $(i, 1)$, we know $pred.(i,1).j = \perp$.

Induction ($n > 1$): Assume true for (k, n) and every interval in the past of (k, n). Suppose the event initiating $(k, n+1)$ is a message send, then $M_k^{n+1} = M_k^n$ except for $M_k^{n+1}[k,k] = M_k^n[k,k] + 1$. Since the event is a message send, (k, n) and $(k, n+1)$ have the exact same predecessors. Thus $pred.(i, M_k^{n+1}[i,i]).j$ equals $pred.(i, M_k^n[i,i]).j$, which by induction hypothesis, equals $(j, M_k^n[i,j])$, which is equal to $(j, M_k^{n+1}[i,j])$ since $i \neq j$ and only element $M_k^n[k,k]$ changes.

Suppose the event initiating $(k, n+1)$ is a message receive tagged with matrix clock W. By the induction hypothesis, W satisfies M2. Note also that M2 relates elements in a single row of M, thus by copying rows from W to M, M2 is not violated. Since all rows $i \neq k$ are either copied from W or not changed, then M2 holds for rows $i \neq k$ in M_k^{n+1}. It remains to show that M2 holds for row k of M_k^{n+1}. The predecessor of $(k, n+1)$ on P_j is the greater of $M_k^n[k,j]$ and $W_k^n[j,j]$. This is exactly the value assigned to $M_k^{n+1}[k,j]$. Thus $(k, M_k^{n+1}[k,j]) = pred.(k, M_k^{n+1}[k,k]).j$ and M2 holds for row k.

(M3) **Base** ($n = 1$): True by initial assignment to matrix.

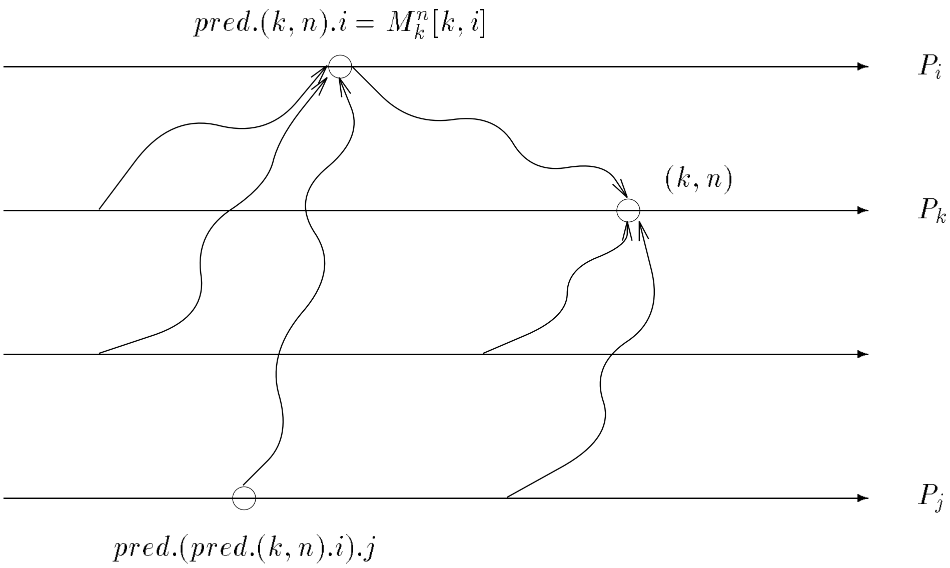
Induction ($n > 1$): Assume $diag(M_k^n) = M_k^n[k, \cdot]$, show $diag(M_k^{n+1}) = M_k^{n+1}[k, \cdot]$. P_k enters interval $(k, n+1)$ only after sending or receiving a message. In the case of a send, $M_k^n[k,k]$ is incremented by one. Thus the diagonal will still be equal to row k. In the case of a message receive, the last statement sets row k to the diagonal.

□

6.4 An application of matrix clocks: Discarding obsolete information

From the above discussion, it can be observed that a matrix of dimension k lets us compute k compositions of *pred* function. Thus, when $k = 0$, we cannot compute pred function. When $k = 1$, the algorithm reduces to the vector clock, and $pred(s).i = s.v[i]$ when $s.p \neq i$. When $k = 2$, we get the two dimensional matrix, and $pred(pred(s).i).j = s.m[i,j]$. This can be extended to any dimension.

Computing pred function multiple times can be useful in many situations. We illustrate one. The following theorem says that if process $s.p$ finds that its $s.p$ column is uniformly bigger than $r.v[s.p, s.p]$ (that is, its local clock in some previous state r), then the information at r has been received by all processes.



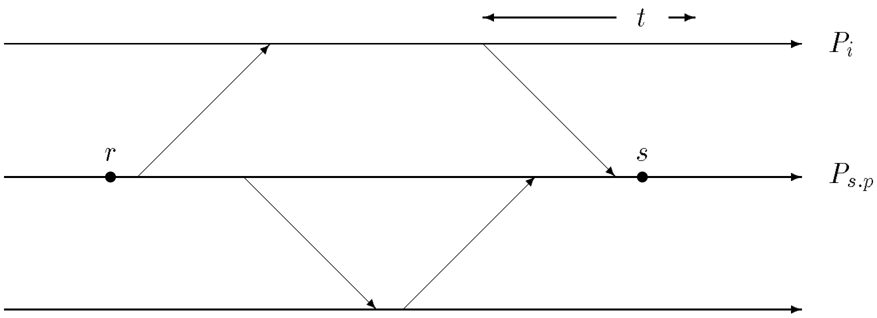
Theorem 21 *Let $s.p = r.p$. If $(\forall i : r.v[r.p, r.p] \leq s.v[i, s.p])$, then $\forall t : t \| s : r \rightarrow t$.*

Sketch of the Proof:

$$
\begin{aligned}
t \| s &\Rightarrow pred(s).i \rightarrow t && (3)\\
(\forall i : r.v[r.p, r.p] \leq s.v[i, s.p]) &\Rightarrow r \preceq pred.(pred(s).i).(s.p) && (4)
\end{aligned}
$$

From the above two equations: $r \rightarrow t$.

■



7 Bibliographic Notes

The most influential paper in this area is by Lamport who first gave the definition of causally precedes [Lam78]. The idea of vector clocks in its pure form first appeared in [Fid89] and [Mat89]. However, vectors had been used before in some earlier papers such as [SY85]. The inductive proof of vector clock based on $\not\rightarrow$ first appeared in [GT93]. In [CB91], it has been shown that the dimension of a vector clock must be at least n (the number of processes) for it to characterize concurrency of a distributed system. The use of vector clocks in algorithms for distributed debugging[Fid89, GW94, TG93], distributed simulation[Mat93], distributed mutual exclusion[Lam78, GT94], and distributed recovery[SY85, PK93] is now well-known. A survey of clock algorithms without proofs also appears as [Ray92].

References

[CB91] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[GT93] V. K. Garg and A. I. Tomlinson. Using induction to prove properties of distributed programs. In *Proc. of the* 5^{th} *IEEE Symposium on Parallel and Distributed Processing*, pages 478–485, Dallas, TX, December 1993. IEEE.

[GT94] V. K. Garg and A. I. Tomlinson. Causality versus time: How to specify and verify distributed programs. In *Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 249 – 256, Dallas, TX, October 1994.

[GW94] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Mat93] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, pages 423–434, August 1993.

[PK93] S. L. Peterson and P. Kearns. Rollback based on vector time. In *Proc. 12th IEEE Symposium on Reliable Distributed Systems*, October 1993.

[Ray92] M. Raynal. About logical clocks for distributed systems. *ACM Operating Systems Review*, 26(1):41–48, 1992.

[SY85] R. E. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[TG93] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, CA, May 1993. ACM/ONR.