Virtual Time and Global States of Distributed Systems [*]

Friedemann Mattern [†]

Department of Computer Science, University of Kaiserslautem
D 6750 Kaiserslautern, Germany

Abstract

A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. However, the notion of time is an important concept in every day life of our decentralized "real world" and helps to solve problems like getting a consistent population census or determining the potential causality between events. We argue that a linearly ordered structure of time is not (always) adequate for distributed systems and propose a generalized non-standard model of time which consists of vectors of clocks. These clock-vectors are partially ordered and form a lattice. By using timestamps and a simple clock update mechanism the structure of causality is represented in an isomorphic way. The new model of time has a close analogy to Minkowski's relativistic space-time and leads among others to an interesting characterization of the global state problem. Finally, we present a new algorithm to compute a consistent global snapshot of a distributed system where messages may be received out of order.

1 Introduction

An *asynchronous distributed system* consists of several processes without common memory which communicate solely via messages with unpredictable (but non-zero) transmission delays. In such a system the notions of *global time* and *global state* play an important role but are hard to realize, at first sight even their definition is not all clear. Since in general no process in the system has an immediate and complete view of all process states, a process can only *approximate* the global view of an idealized external observer having immediate access to all processes.

The fact that *a priori* no process has a consistent view of the global state and a common time base does not exist is the cause for most typical problems of distributed systems. Control tasks of operating systems and database systems like *mutual exclusion, deadlock detection, and concurrency control* are much more difficult to solve in a distributed environment than in a classical centralized environment, and a rather large number of distributed control algorithms for those problems has found to be wrong. *New problems* which do not exist in centralized systems or in parallel systems with common memory also emerge in distributed systems. Among the most prominent of these problems are *distributed agreement, distributed termination detection*, and the *symmetry breaking* or *election problem*. The great diversity of the solutions to these problems—some of them being really beautiful and elegant—is truly amazing and exemplifies many principles of distributed computing to cope with the absence of global state and time.

Since the design, verification, and analysis of algorithms for asynchronous systems is difficult and error-prone, one can try to

1. simulate a *synchronous* distributed system on a given asynchronous systems,
2. simulate global time (i.e., a common clock),
3. simulate global state (i.e., common memory),

and use these simulated properties to design simpler algorithms. The first approach is realized by so-called *synchronizers* [1] which simulate clock pulses in such a way that a message is only generated at a clock pulse and will be received before the next pulse. A synchronizer is intended to be used as an additional layer of software, transparent to the user, on top of an asynchronous system so that it can execute synchronous algorithms. However, the message overhead of this mechanism is rather high. The second approach does not need additional messages and the system remains asynchronous in the sense that messages have unpredictable

[*] This is an edited version, with minor alterations, of the paper with same title which originally appeared in the *Proceedings of the International Workshop on Parallel and Distributed Algorithms* (Chateau de Bonas, France, October 1988), M. Corsnard et al. (ed.) copyright 1989 Elsevier Science Publishers B. V. (North-Holland), with author's permission.

[†] Current address: Department of Computer Science, University of Saarland, 6600 Saarbrucken, Germany. Email address: mattern@cs.uni-sb.de

transmission delays. This approach has been advocated by Lamport [9]. He shows how the use of *virtual time* implemented by logical clocks can simplify the design of a distributed mutual exclusion algorithm. Morgan [11] and Neiger and Toueg [12] develop this idea further. The last approach is pursued by Chandy and Lamport in their *snapshot algorithm* [3], one of the fundamental paradigms of distributed computing. Panangaden and Taylor [14] elaborate the idea leading to the characterization of "concurrent common knowledge".

Obviously, the notions of global time and global state are closely related. By Chandy and Lamport's algorithm a process can, without "freezing" the whole computation, compute the "best possible approximation" of a global state—a global state that *could* have occurred if all processes took a snapshot of their local state simultaneously. Although no process *in* the system can decide whether the snapshot state did really occur, "could have occurred" is good enough for stable properties (i.e., global predicates which remain true once they become true) like termination detection and deadlock detection, which shows that the snapshot algorithm is a general solution to these problems. (This does not depreciate the specific algorithms for those problems, however, which are often simpler and more efficient). While in some sense the snapshot algorithm computes the best possible attainable global state approximation, Lamport's virtual time algorithm is not that perfect. In fact, by mapping the partially ordered events of a distributed computation onto a linearly ordered set of integers it is *losing information.* Events which may happen simultaneously may get different timestamps as if they happen in some definite order. For some applications (like mutual exclusion as described by Lamport himself in [9]) this defect is not noticeable. For other purposes (e.g., distributed debugging), however, this is an important defect.

In this paper, we aim at improving Lamport's virtual time concept. We argue that a linearly ordered structure of time is not always adequate for distributed systems and that a partially ordered system of *vectors* forming a lattice structure is a natural representation of time in a distributed system. This *non-standard model of time* resembles in many respects Minkowski's relativistic space-time. In particular, it has an extended range of "simultaneity"—all events which are not causally related are simultaneous—thus representing causality in an isomorphic way without loss of information. The new notion of time together with a generalized clock synchronization algorithm yields for every process the best approximate knowledge of the "idealized" time of an external observer. (This idealized global time is quite naturally defined as the supremum of all local clock vectors). The vector structure of time will be shown to be isomorphic to the structure of possible states as constructed by the snapshot algorithm, thus yielding a complementary view of global time and global state.

2 Event structures

In an abstract setting, a process can be viewed as consisting of a sequence of *events*, where an event is an atomic transition of the local state which happens in no time. Hence, events are *atomic actions* which occur at processes. Usually, events are classified into three types: send events, receive events, and internal events. An *internal event* only causes a change of state. A *send event* causes a message to be sent, and a *receive event* causes a message to be received and the local state to be updated by the values of the message. Notice, however, that in so-called *message* − *driven* models of distributed computing (e.g., the *actor model*) there is only one type of event: The receipt of a message triggers the execution of an atomic action resulting in a local state update and in any finite number of messages sent to other processes. Because of their simplicity, message-driven models are attractive from an abstract point of view.

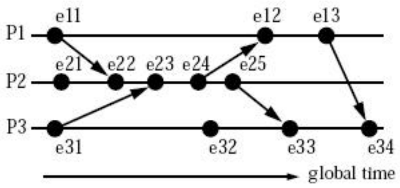

Fig. 1. An event diagram

Events are *related*: Events occurring at a particular process are totally ordered by their local sequence of occurrence, and each receive event has a corresponding send event. This relationship is the heart of any notion of *virtual time.* However, the central concept seems to be the *causality relation* which determines the primary characteristic of time, namely that the *future cannot influence the past.* Formally, an *event structure* [13] is a pair $(E, <)$, where E is a set of events, and '$<$' is an irreflexive partial order on E called the *causality relation.* Event structures represent distributed computations in an abstract way. For a given computation, $e < e$' holds if one of the following conditions holds:

(1) e and e' are events in the same process and e precedes e',

(2) e is the sending event of a message and e' the corresponding receive event,

(3) $\exists e''$ such that $e < e''$ and $e'' < e'$.

The causality relation is the smallest relation satisfying these conditions.

It is helpful to view this definition in terms of a diagram (Figure 1). Obviously, $e < e'$ signifies that it is *possible* for event e to causally affect event e'. Graphically, this means that one can follow a "*path of causality*" from event e to event e' in the diagram (moving in the direction of the arrows and from left to right on the process lines).

It is possible to view a diagram like Figure 1 as a *timing diagram* of an actual computation where the horizontal direction represents *real time*. Then the events depicted by dots occur at some specific instant of time as observed by an idealized external observer. Messages are represented by diagonal arrows. (An observer who can continuously watch the messages on their way to their destination could also draw an even more accurate diagram of the actual message flow. Notice that a two-dimensional space-time diagram represents the loci of objects of a $one-dimensional$ space against time). It is also possible to view the diagram only as an abstract $poset-diagram$ of the partial order $(E,<)$. Usually, poset-diagrams are drawn by placing an element e higher than e' whenever $e' < e$. (Because transitivity is assumed, only the connections of directly related elements are drawn, redundant connections are omitted). Figure 2 shows the poset diagram for the computation of Figure 1. Obviously, the two diagrams are isomorphic. However, Figure 1 seems to make implicit allusions to global time depicting a specific computation, whereas Figure 2 only shows the *logical* relationships of events, i.e., the *causal structure* of the computation.

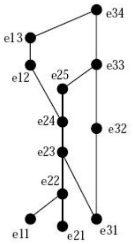

Fig. 2. The poset diagram

Figure 3 shows a diagram which is very similar to the diagram depicted in Figure 1. Notice that the partial event order is the same (i.e., it has the same poset-diagram), but that here events e_{12}, e_{25}, and e_{32} happen at the same global time instant. Such diagrams, showing the same causality relation, will be called *equivalent*.

Obviously, a time diagram can be transformed to another equivalent diagram by stretching and compressing the horizontal process lines representing the local time axis. One gets an operational view of the *equivalence transformation on time diagrams* by assuming that the process lines consist of idealized rubber bands. Any time diagram d' which can be constructed out of a given time diagram d by stretching and compressing the elastic bands is equivalent to d, as long as the arrows representing message flow do not go backwards (i.e., from the right to the left) in time. (Time diagrams with messages flowing backwards in time obviously do not depict any realizable situation. In any "valid" time diagram an event e must be drawn to the left of an event e' whenever $e < e'$—this is the only restriction when going from a poset-diagram to an actual time diagram). Notice that the "rubber band equivalence transformations" on time diagrams are exactly those transformations which *leave the causality relation invariant.*

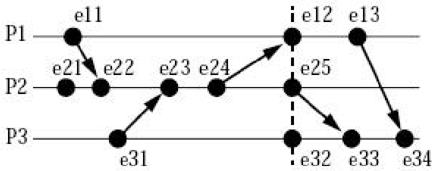

Fig. 3. An equivalent diagram

3 Consistent cuts

If a process sends messages to all other processes in order to initiate local actions or to recall some distributed information, these messages will usually be received at different time instants. Due to unpredictable message delays it is not possible to guarantee that all local actions triggered by the messages are performed simultaneously. This motivates the notion of "cuts".

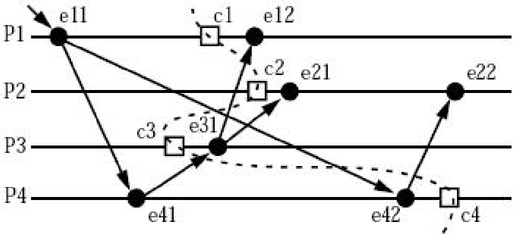

Fig. 4. A cut

Graphically, a *cut* (or *time slice*) is a zigzag line cutting a time diagram into two parts—a left part and a right part (Figure 4). Formally, we augment the set of events E to include a new so-called cut event c_i for each process P_i : $E' = E \cup \{c_1,\ldots,c_n\}$ (n denotes the number of processes). Connecting these events in the diagram yields the *cut line*. A cut partitions E into two sets PAST (those events that happen before the cut) and FUTURE (those events that happen after the cut) [2]. Let $<_l$ denote the local event order (i.e., $e <_l e'$ iff $e < e'$ and e and e' occur at the same process). Then we can formally identify a cut with its PAST set yielding the following definition:

Definition 1 *A cut C of an event set E is a finite subset $C \subseteq E$ such that $e \in C\ \ \&\ \ e' <_l e \rightarrow e' \in C$.*

Here we have lost the cut events $c_1, \ldots, c_n$. However, it is straightforward to associate "the" cut as defined by Definition 1 to a given set of cut events or vice versa. It will be clear from the context whether we mean by a cut the PAST set or the cut events. Figure 5 shows the poset-diagram with the set PAST for the cut depicted in Figure 4.

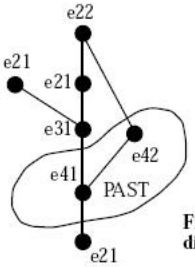

Fig. 5. The poset diagram of Fig. 4.

The following definition already anticipates the notion of time:

Definition 2 *A cut C_1 is later than cut C_2 if $C_1 \supseteq C_2$.*

In the diagram, a cut line of a cut C_1 being later than a cut C_2 is to the right of C_2's cut line. (In Figure 6, C_3 is later than C_1 and later than C_2. However, neither is C_1 later than C_2 nor vice versa). Notice that "later than" is reflexive, i.e., a cut is later than itself. "Later than" is a *partial order* on the set of cuts. Moreover, it forms a *lattice*:

Theorem 1 *With operations $\cup$ and $\cap$ the set of cuts of a partially ordered event set E forms a lattice.*

The proof is straightforward. Recall that a lattice is a partially ordered set any two of whose elements have a greatest lower bound *inf* and a least upper bound *sup*. Obviously, $inf = C_1 \cap C_2$ and $sup = C_1 \cup C_2$ for any two cuts C_1, C_2.

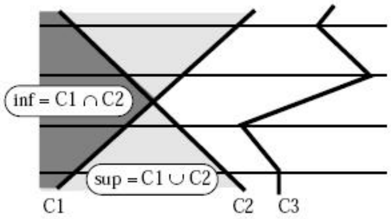

Fig. 6. The lattice structure of cuts

If no provisions are taken, it can happen that a cut contains the receiving event of a message, but not its sending event. Such a situation is undesirable because cuts are used to compute the global state of a distributed system along a cut line (Section 4). The following definition rules out such inconsistent cuts by requesting that cuts are left-closed under the causality relation '$<$':

Definition 3 *A consistent cut C of an event set E is a finite subset $C \subseteq E$ such that $e \in C\ \ \&\ \ e' < e \rightarrow e' \in C$.*

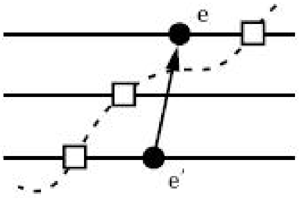

Fig. 7. An inconsistent cut

A cut is consistent if every message received was sent (but not necessarily vice versa!). Figure 4 shows a consistent cut. The cut of Figure 7 is inconsistent: event e is part of the cut, but its immediate predecessor e' is not. Obviously, because $e <_l e'$, the set of consistent cuts is a subset of the set of all cuts of some partially ordered set E. But that is not all:

Theorem 2 *The set of consistent cuts is a sublattice of the set of all cuts.*

The simple proof that the consistent cuts are closed under $\cup$ (*sup*) and $\cap$ (*inf*) is left to the reader.

The lattice structure of consistent cuts guarantees that for any two consistent cuts C_1, C_2 there is always a cut later than both of them and a cut earlier than both of them. This extends to any finite set of consistent cuts: $sup(C_1, \ldots, C_k) = C_1 \cup \ldots \cup C_k$ is later than $C_1, \ldots, C_k$.

The following two theorems are stated without proofs (but see [14] and [3]). The graphical interpretation is obvious. Assume that a cut line is also a rubber band. Now stretch that band so that it becomes straight vertical. If then a message arrow crosses it from the right to the left, the cut is inconsistent. Otherwise it is consistent. Figure 8 shows the "rubber band consistency test" for an inconsistent cut. (Notice that $c_3 < e' < e < c_1$ and hence $c_3 < c_1$, thus violating the condition of Theorem 3).

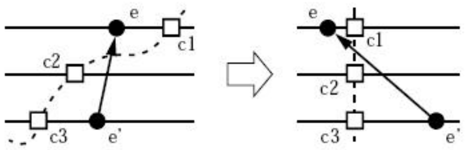

Fig. 8. The rubber band test

Theorem 3 *For a consistent cut consisting of cut-events $c_1, \ldots, c_n$, no pair of cut-events is causally re-*

lated, i.e., $\neg(c_i < c_j)$ & $\neg(c_j < c_i)$ for all cut events c_i, c_j.

By taking into consideration the remarks at the end of the previous section one sees that consistent cuts are "possible":

Theorem 4 *For any time diagram with a consistent cut consisting of cut-events $c_1, \ldots, c_n$, there is an equivalent time diagram where $c_1, \ldots, c_n$ occur simultaneously, i.e., where the cut line forms a straight vertical line.*

4 Global states of consistent cuts

As Theorem 4 shows, all cut-events of a consistent cut can occur simultaneously, i.e., there is a potential execution of the distributed computation in which all cut-events are indeed simultaneous in "real time". A snapshot of the local states of all processes taken at the same (real) time is obviously consistent. Therefore, the global state computed along a consistent cut is "correct".

The *global state* of a consistent cut comprises the local state of each process at the time the cut-event happens and the set of all messages sent but not yet received. In the time diagram these messages are characterized by arrows crossing the cut line from the left to the right side.

The *snapshot problem* consists in designing an efficient protocol which yields only consistent cuts and to collect the local state information. Furthermore, in some way the messages crossing the cut must be captured. Chandy and Lamport presented such an algorithm for the first time assuming that message transmission is FIFO [3]. We propose a similar algorithm in Section 11 for non-FIFO channels.

5 The concept of time

As Lamport notes, the concept of time is fundamental to our way of thinking [9]. In fact, "real time" helps to master many problems of our decentralized real world. Consider, for example, the problem of getting a consistent population census. Here one agrees upon a common time instant (being far enough in the future) and gets everyone counted at the same moment. Time is also a useful concept when considering possible causality. Consider a person suspect of a crime, if that person has an alibi because he or she was far enough away from the site of the crime at some instant close enough to the time of the crime, then he or she cannot be the culprit (Figure 9).

The examples work because of the characteristics of real time. Using clocks, events e, e' can be timestamped by $t(e)$ and $t(e')$ such that whenever e happens earlier than e' and might thus causally affect e', then $t(e) < t(e')$. According to Lamport it is one of the "mysteries of the universe" that it is possible to construct a system of clocks which, running independently of each other, observe the causality relation.

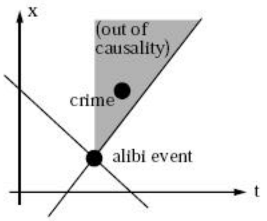

Fig. 9. Time and causality

In asynchronous distributed systems without real time clocks a common time base does not exist. But it would be fine if we had an approximation having most of the features of real time—the design of distributed algorithms would be considerably simplified. The idea of Morgan [11] is to factorize distributed algorithms in two separate algorithms:

(1) An algorithm where global time is available to all processes.

(2) A clock synchronization algorithm realizing *virtual time* as a suitable approximation of global time.

This separation of concern helps to design new distributed algorithms: once the idea has been found and elaborated it should be possible to combine the two components into a single optimized algorithm.

However, these statements are rather vague and give raise to a few questions: What exactly is virtual time, and what should be considered to be a "best possible approximation" of global (i.e., real) time? And given an algorithm which was written assuming the existence of real time, is it still correct when virtual time is used? What is the essential structure of real time? These questions will be considered in the sequel.

The nature of time is studied among others by philosophers, mathematicians, physicists, and logicians. It is its formal structure in the model-theoretic sense which is of interest for us, and here the prevalent mathematical picture of "standard time" is that of a set of "instants" with a temporal precedence order '<' ("earlier than") satisfying certain obvious conditions [17]:

(1) *Transitivity.*

(2) *Irreflexivity* (notice that transitivity and irreflexivity imply *asymmetry*).

(3) *Linearity.*

(4) *Eternity* ($\forall x \exists y : y < x, \forall x \exists y : x < y$).

(5) *Density* ($\forall x, y : x < y \rightarrow \exists z : x < z < y$).

There are several non-isomorphic models satisfying these axioms, the most obvious are the rationales Q and the reals R. But in most cases when using real time and clocks we do not need all these properties—for example digital clocks obviously do not satisfy the density axiom but are nevertheless useful in many cases. (When replacing density by

(5') *discreteness*

the integers Z are the standard model of time).

This shows that implementing clocks by hardware counters or by variables of type real or integer in computer programs is indeed "correct". Occasionally, however, we notice that in fact axioms (4) and (5) are not perfectly realized: In large simulation programs for example it may happen that the clock variable overflows (simulation time is not eternal) or that rounding errors result in events happening at the same time when they shouldn't (simulation time is not dense).

The main question is: Can we design a system of logical clocks and a synchronization mechanism for asynchronous distributed systems which fulfills axioms (1) to (5) (or (5')) without making use of real time or physical clocks or similar mechanisms? And can we use that system of logical clocks to timestamp events such that the causality relation is preserved? As it turns out, Lamport's clock synchronization mechanism [9] which we will sketch in the next section fulfills these requirements and is useful in many respects. Despite that, however, it has a defect because it does not preserve causal *independence*. As we will see, preserving causal independence is possible by using N^n, Z^n, Q^n, or R^n (n denoting the number of processes) as the "time domain".

6 Virtual time

The main difference between virtual time and real time seems to be that virtual time is only identifiable by the succession of events (and therefore is discrete). Virtual time does not "flow" by its own means like real time whose passage we can't escape or influence. Just doing nothing and waiting for the virtual time to pass is dangerous—if nothing happens virtual time stands still and the virtual time instant we are waiting for may never occur. The concept of virtual time for distributed systems was brought into prominence by Lamport in 1978 [9]. It is widely used in distributed control algorithms (but not always made explicit), e.g., in mutual exclusion algorithms and concurrency control algorithms. Morgan gives some applications of virtual time [11] and Raynal shows that the drift between logical clocks in different processes can be bounded [16].

A *logical clock* C is some abstract mechanism which assigns to any event $e \in E$ the value $C(e)$ (the *timestamp* of e) of some "time domain" T such that certain conditions are met. Formally, a logical clock is a function $C : E \rightarrow T$, where T is a partially ordered set such that the *clock condition*

$$e < e' \rightarrow C(e) < C(e')$$

holds. The irreflexive relation '<' on T is called "earlier than", its converse "later than". Stated verbally, the clock condition reads *if an event e can causally affect another event e', then e has an earlier (i.e., smaller) timestamp than e'.* Notice that the converse implication is not required.

As a consequence of the clock condition the following properties hold [11]:

(1) If an event e occurs before event e' at some single process, then event e is assigned a logical time earlier than the logical time assigned to event e'.

(2) For any message sent from one process to another, the logical time of the send event is always earlier than the logical time of the receive event.

Usually, the set of integers N is taken for the time domain T, and the logical clock is implemented by a system of counters C_i, one for each process P_i. To guarantee the clock condition, the local clocks must obey a simple protocol:

(1) When executing an internal event or a send event at process P_i the clock C_i "ticks":

$$C_i := C_i + d \qquad (d > 0).$$

(2) Each message contains a timestamp which equals the time of the send event.

(3) When executing a receive event at P_i where a message with timestamp t is received, the clock is advanced:

$$C_i := max(C_i, t) + d \qquad (d > 0).$$

We assume that the timestamp of an event is already the new value of the local clock (i.e., updating the local clock occurs just before executing the event). A typical value of d is $d = 1$, but d can be different for each "tick" and could also be some approximation of the real time difference since the last "tick".

Two events e, e' are mutually independent (denoted by $e \parallel e'$) if $\neg(e < e')$ & $\neg(e' < e)$. Can we say anything about the timestamps of mutually independent events? Figure 10 shows that events which are causally independent may get the same or different timestamps. (Here it is assumed that initially $C_i = 0$ and $d = 1$). e_{12} and e_{22} have the same timestamp, but e_{11} and e_{22} or e_{13} and e_{22} have different values.

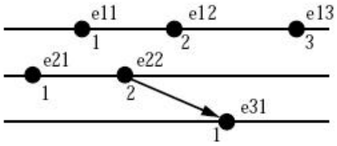

Fig. 10. Timestamps of events

Figure 11 depicts this situation. Looking at their timestamps, we can conclude that two events are independent if they have the same timestamps. We can also conclude that if $C(e) < C(e')$ then $\neg(e' < e)$, i.e., it is guaranteed that the past cannot be influenced by the future. However, if $C(e) < C(e')$ it is *not* possible to decide whether $e < e'$ or $e \parallel e'$, i.e., whether the events are causally related or not. (Notice that the causal relation $e < e'$ does not necessarily mean that e causes e' in some modal sense. It only means that e *could* cause e'). This is an important defect—by looking at the timestamps of events it is not possible to assert that some event could *not* influence some other event.

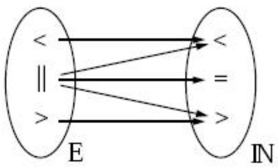

Fig. 11. Loss of structure

The reason for the defect is that C is an order-homomorphism which preserves '<' but which obliterates a lot of structure by mapping E onto a linear order—it does not preserve negations (e.g., "||"). What we are looking for is a better suited domain set for T and an *isomorphism* mapping E onto T.

7 Vector time

Assume that each process has a simple clock C_i which is incremented by 1 each time an event happens. An idealized external observer having immediate access to all local clocks knows at any moment the local times of all processes. An appropriate structure to store this global time knowledge is a vector with one element for each process. The example depicted in Figure 12 illustrates the idea.

Our aim is to construct a mechanism by which each process gets an *optimal approximation* of this global time. For that, we equip each process P_i with a clock C_i consisting of a *vector* of length n, where n is the total number of processes. (Since time will henceforth be considered to be a set of vectors, it is consequent that clocks, i.e., devices which hold the time, are implemented by vectors or "arrays"). With each event the local clock "ticks". A process P_i ticks by incrementing its own component of its clock:

$$C_i[i] := C_i[i] + 1.$$

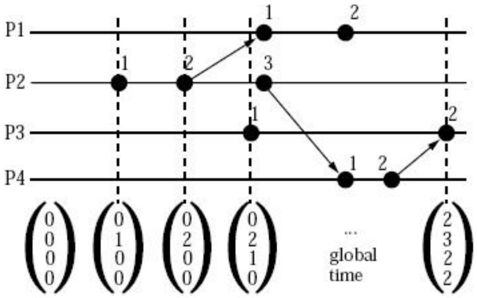

Fig. 12. Global vector time

(Any real-valued increment $d > 0$ instead of 1 is acceptable). As before, ticking is considered to occur before any other event action, and the *timestamp C(e) of an event e* is the clock value after ticking. (However, notice that now timestamps are vectors). As in Lamport's original scheme each message gets a piggybacked timestamp consisting of the vector of the local clock. By receiving a timestamped message a process gets some knowledge about the other processes' global time approximation. The receiver combines its own knowledge about global time (C_i) and the approximation it receives (t) simply by

$$C_i := sup(C_i, t)$$

where t is the timestamp of the message and *sup* is the componentwise maximum operation, i.e., $sup(u, v) = w$ such that $w[i] = max(u[i], v[i])$ for each component i. Because of the transitivity of the scheme, a process may also receive time updates about clocks in non-neighboring processes. Figure 13 shows an example of the time propagation scheme.

Causally independent events may happen in any order. Moreover, it is somewhat arbitrary whether in Figure 13 global time is incremented by (0, 0, 0, 0), (0, 1, 0, 0), (0, 1, 0, 1) or by (0, 0, 0, 0), (0, 0, 0, 1), (0, 1, 0, 1)—both sequence are possible and reflect the flow of time for different but equivalent time diagrams. However, time (1, 3, 0, 0) can never be observed for the computation depicted in Figure 13, it is an "impossible" clock value.

Notice that since only process P_i can advance the i-th component of global time, it always has the most accurate knowledge of its own local time. This yields the following theorem:

Theorem 5 *At any instant of real time* $\forall i, j : C_i[i] \geq C_j[i]$.

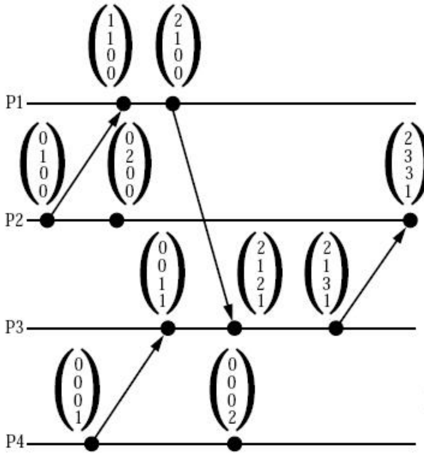

Fig. 13. Time propagation

It is possible to compare time vectors by defining the relations '$\leq$', '$<$', and '$||$':

Definition 4 *For two time vectors u, v*

$u \leq v$ *iff* $\forall i : u[i] \leq v[i]$.

$u < v$ *iff* $u \leq v$ & $u \neq v$, *and*

$u \,||\, v$ *iff* $\neg(u < v)$ & $\neg(v < u)$.

Notice that '$\leq$' and '$<$' are partial orders. The reflexive and symmetric *concurrency relation* '$||$' is a generalization of the *simultaneity* relation of standard time. Whereas the notion of "present" in standard time is merely an extensionless intersection point between past and future, here we have a larger range. Notice, however, that the concurrency relation is not transitive!

The following definition assigns a time to a cut:

Definition 5 *Let X be a cut and* c_i *denote the cut event of process* P_i *(or the maximal event of* P_i *belonging to X if no special cut events exist—recall that we assume the existence of an initial event for each process). Then*

$$t_X = sup(C(c_1), \ldots, C(c_n))$$

is called the global time of cut X. *(Notice that sup is associative).*

As Figure 14 shows, different cuts can have the same time ((2, 1, 2, 0) in the example). However, for *consistent* cuts the time is unique yielding a practical *consistency criterion* for cuts:

Theorem 6 *Let X and* c_i *be defined as in Definition 5. X is consistent iff* $t_X = (C(c_1)[1], \ldots, C(c_n)[n])$.

Proof:

(1) If X is consistent then by Theorem 4 we can assume that $c_1, \ldots, c_n$ happen at the same instant of real time. Then Theorem 5 applies and yields the result.

(2) If X is inconsistent then there exists a message sent after c_i by some process P_i and received before c_j by another process P_j. If t denotes the timestamp of the message then $c_i[i] < t[i] \leq c_j[i]$. Therefore $t_X = (C(c_1)[1], \ldots, C(c_n)[n])$.

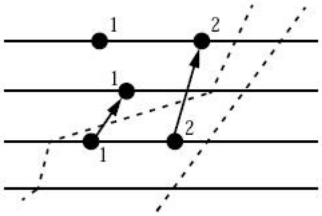

Fig. 14. Time of cuts

8 The structure of vector time

The non-linear vector time has an interesting structure. First we have the following theorem:

Theorem 7 *For any* $n > 0$, $(N^n, \leq)$ *is a lattice.*

The proof is obvious. For any two vectors $u, v \in N^n$ the least upper bound is simply $sup(u, v)$ and the greatest lower bound is $inf(u, v)$, where inf is defined analogously to sup by replacing max by min. (Instead of N^n, we can use Z^n, Q^n or R^n). Things become more interesting if we consider only the *possible* time vectors of an event set E. In analogy to Theorem 2 one can prove

Theorem 8 *The set of possible time vectors of an event set E is a sublattice of* $(N^n, \leq)$.

In fact, it is possible to "identify" a consistent cut X with its time vector t_X. The identification of possible time vectors and consistent cuts leads to the following main theorem:

Theorem 9 *For an event set E, the lattice of consistent cuts and the lattice of possible time vectors are isomorphic.*

This is the isomorphism we were looking for! Again, we leave the proof to the reader. Theorem 9 has a nice and important consequence:

Theorem 10 $\forall e, e' \in E : e < e'$ *iff* $C(e) < C(e')$ *and* $e \,||\, e'$ *iff* $C(e) \,||\, C(e')$.

This gives us a very simple method to decide whether two events e, e' are causally related or not: We take their timestamps C(e) and C(e') and check whether $C(e) < C(e')$ or $C(e') < C(e)$. If the test succeeds, the events are causally related. Otherwise they are causally independent. The test can be simplified if the processes where the events occur are known:

Theorem 11 *If e occurs at process P_i then for any event $e' \neq e : e < e'$ iff $C(e)[i] \leq C(e')[i]$.*

¿From the pictorial meaning of the causality relation the correctness of Theorem 10 and Theorem 11 is easy to see. If event e can causally affect event e', then there must exist a path of causality in the time diagram which propagates the (local) time knowledge of event e to event e'. Propagating time knowledge t along a path can only *increase* t. Conversely, if event e' "knows" the local time of e, there must exist a causality path from e to e'. (See also [6] and [5]).

Any path of the poset-diagram corresponds to a possible *interleaving*—a linear sequence of events which is consistent with the causality relation. Therefore the set of paths determines the *possible development of global virtual time*. However, time cannot always spread freely in all dimensions, messages put *restrictions* on the development of time—a message sent by process P_i when $C_i[i] = s$ and received by process P_j when $C_j[j] = r$ induces the restriction $C[i] < s \rightarrow C[j] < r$ or equivalently $\neg(C[i] < s \;\&\; C[j] \geq r)$.

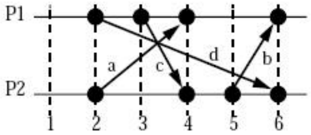

Fig. 15. A diagram with four messages

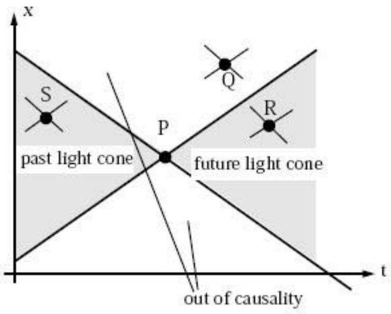

Fig. 17. Light cones

The time diagram depicted in Figure 15 shows four messages, Figure 16 displays the lattice structure together with the area cut off by the four restrictions $\neg(C[2] < 2 \;\&\; C[1] \geq 4), \neg(C[2] < 5 \;\&\; C[1] \geq 6), \neg(C[1] < 3 \;\&\; C[2] \geq 4), \neg(C[1] < 2 \;\&\; C[2] \geq 6)$. (The lattice structure of systems of three processes can be esthetically more pleasing.)

9 Minkowski's space-time

The standard model of time fulfills axioms (1)-(5) of Section 5. However, reality is not "standard". In fact, because of the finite speed of light, Minkowski's well-known relativistic space-time model may reflect reality more accurately than the standard model of time. It has some nice analogies to our vector model of time.

In Minkowski's model n-dimensional space and one-dimensional time are combined together to give a $n+1$-dimensional picture of the world. While in physical reality $n = 3$, we restrict ourselves to the case $n = 1$ giving a two dimensional structure (i.e., a *time diagram*) with one temporal and one spatial dimension. Due to the finite velocity of signals, an event can only influence events which lie in the interior of the so-called *future light cone* as depicted in Figure 17. (Notice the similarity to Figure 9, both describe the same phenomenon). Events P and Q are "out of causality", whereas S can influence P, and P can influence R. Given the two coordinates of events e and e' (and the speed of light) it is a simple arithmetic exercise to check whether the events are out of causality or which event may influence the other.

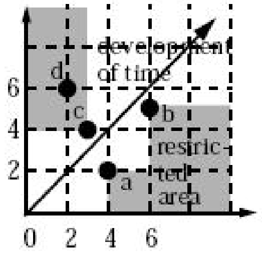

Fig. 16. Restrictions

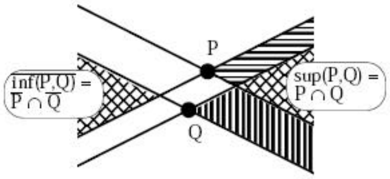

Fig. 18. Lattice structure of light cones

Notice that we have a *partially ordered* set of events, and each event has a "space-time stamp" (its coordinates). The *causality preserving transformations* (which correspond to our "rubber band transformations") play also an important role in Minkowski's space-time, these are the famous *Lorentz transformations* leaving the light cone invariant. The analogy goes even further—the light cones of the two-dimensional Minkowski space form a *lattice*. Figure 18 depicts the construction. We simply identify a space-time point with its light cone. The intersection of the future light cones define the supremum, whereas the intersection of the past light cones define the infimum. (Notice that any two light cones intersect). This identification is analogous to our identification of cuts and time vectors.

At first sight, the strong structural analogy between Minkowski's space-time and our vector time is surprising—it seems to indicate that vector time is based on a fundamental and non-fictitious concept. However, the real world *is* distributed and the analogy might not be as strong as it seems: The *three dimensional* light cones do *not* form a lattice!

10 Application of vector time

An interesting application of the vector concept is the field of *distributed debugging*. To locate a bug that caused an error, the programmer must think about the causal relationship between events in a program's execution. However, debugging in a distributed environment is more difficult than traditional debugging, mainly because of the increased complexity and unreproducible race conditions. Tracing of events is therefore important, and timestamped trace data can be used to detect possible race conditions. Obviously, a potential *race condition* exists if there is no causal relationship between two events. This can be detected by comparing the timestamps of the vectors. Time vectors can also help in proving that some event cannot be the cause for another event, thus helping to locate an error. (Notice, however, that if $e < e'$, it is only *possible* that e influences e').

Our concept of vector time has been implemented and integrated in a distributed debugging system on our experimental multicomputer system [6]. Independently of us the use of vectors of logical clocks for distributed debugging has recently been suggested and discussed by Fidge [5, 4]. A potential application similar to distributed debugging are so-called *trace checkers* for distributed systems which observe a sequence of events and check whether it is a possible sequence of events according to a given specification [7].

For the purpose of *performance analysis* it is useful to get information about *potential concurrency*. Two events e, e' for which $e \parallel e'$ can be executed concurrently. An analysis of the timestamped trace data of an execution can therefore help to determine the degree of parallelism.

A scheme very similar to our time vectors can be used to *detect mutual inconsistencies of multiple file copies* in the context of network partitioning. In [15] Parker et al. show how vectors can be used to detect situations in which copies of a file were modified independently in separate partitions. Each copy of a file has a so-called *version vector* whose i-th component counts the number of updates made to the file at site i. If $u \parallel v$ (and $u \neq v$) for two version vectors u, v then a *version conflict* is signalled. Two file copies with version vectors u, v which are not in conflict may be reconciled without any further action, the version vector of the reconciled file is $sup(u, v)$.

In a *distributed simulation system* each process has its own logical clock. However, these clocks are not completely unrelated. A process can only safely advance its clock if it is guaranteed that it will not receive a message from some other process which is behind in time. In fact, things are a little bit more complicated, but the main point is that in order to get a high degree of parallelism it is essential that each process has a good approximation of the other processes' logical clocks. Here the concept of vector time applies quite naturally and piggybacking time vector on application messages increases the speedup. A distributed simulation system using this idea is currently being implemented.

Another use of vector time is the *design of distributed algorithms*. Having in some sense the best possible approximation of global time should simplify the development of distributed algorithms and protocols. We will present one such application in the next section.

11 Computing global states on systems without FIFO channels

In this section we "reinvent" a variant of the Chandy-Lamport algorithm using time vectors and show how snapshots can be computed when messages are not necessarily received in the order sent. First, we concentrate on the local states of the processes, the messages in transit will be considered subsequently.

Clearly, the notion of global state of a distributed system is only meaningful for *consistent* cuts. Theorem 6 shows how it can be determined whether a given cut is consistent or not. However, what we are looking for is a method which *guarantees* to yield only consistent cuts.

In the real world the snapshot algorithm (e.g., to obtain a consistent population census) is simple:

(1) All "processes" agree on some "future" time s.

(2) A process takes its local snapshot at time s.

(3) After time s the local snapshots are collected to construct a global snapshot.

If we want to adapt the algorithm for use in distributed systems without a common clock, then the processes have to agree on a single future *virtual* time s or on a set of virtual time instant $s_1, \ldots, s_n$ which are mutually concurrent and did not yet occur. The following theorem is helpful.

Theorem 12 *At the moment clock C_i ticks: $\neg\ \exists j : C_i < C_j$.*

Proof (sketch): Use Theorem 5 and recall that message transmission times are assumed to be non-zero.

This means that it is never too late to request a snapshot "now"; the clocks of other processes cannot be later than P_i's own clock when executing a local event!

For simplicity of exposition we assume that P_i is the only initiator requesting a snapshot. The first idea is as follows:

(1) P_i "ticks" and then fixes its "next" time $s = C_i + (0, \ldots, 0, 1, 0, \ldots, 0)$ to be the common snapshot time. (The "1" is on position i).

(2) P_i broadcasts s to all other processes.

(3) P_i does not execute any event until it knows that every process knows s (e.g., by getting acknowledgments).

(4) Then P_i "ticks" again (thereby setting C_i to s), takes a local snapshot, and broadcasts a dummy message to all processes. This forces all processes to advance their clocks to a value $\geq s$.

(5) Each process takes a local snapshot and sends it to P_i when its local clock becomes equal to s or jumps from some value smaller than s to a value larger than s.

The drawback that P_i is "frozen" until it has successfully published the common snapshot time s can be overcome by "inventing" a $n + 1$st virtual process whose clock is managed by P_i. P_i is now free to use its own clock in the usual way and can behave as any other process. But then it becomes obvious that the first n components of the vectors are of no use—they can simply be omitted! Because the virtual clock C_{n+1} ticks only when P_i initiates a new run of the snapshot algorithm (which we assume will only happen after completion of a previous run), the first broadcast phase is also unnecessary—the processes already know the next snapshot time! It is also easy to see that instead of using an integer counter for the snapshots a counter modulo 2 (i.e., an alternating boolean state indicator) is sufficient.

If we call the two states *white* ("before snapshot") and *red* ("after snapshot") then we end up with a simple algorithm where each message is either white or red (indicating whether it was sent before or after the snapshot) and each process (which is initially white) becomes red and immediately takes a local snapshot as soon as it receives a red message for the first time. White processes only send white messages and red processes only send red messages. The initiating process becomes red spontaneously and then starts a virtual broadcast algorithm to ensure that eventually all processes become red. To achieve that, it may directly or indirectly send (red) dummy messages to all processes (e.g., by using a virtual ring) or it may flood the network by using a protocol where a process sends dummy messages to all its neighbors when it becomes red.

Notice that the algorithm is correct even if messages are not received in the order sent. It is easy to see that the cut induced by the snapshot algorithm (which consists of all "white events") is *consistent*. There does not exist a message sent by a red process which is received by a white process because such a message would color the receiving process red immediately before the message is accepted.

The full global state of a consistent cut not only consists of the local states but also of the *messages in transit*. Fortunately, those messages are easily identifiable—they are white but are received by a red process. Whenever a red process gets such a message it simply sends a copy of it to the initiator.

The only remaining problem is *termination*. The initiator gets copies of all messages in transit but it does not know when it has received the last one. In principle, the problem can be solved by any *distributed termination detection algorithm* for non-FIFO channels [10] by simply ignoring the red messages. However, a *deficiency counting method* is particularly attractive in this case because a consistent snapshot of the message counters can be taken at the same moment as the snapshot of the local states.

For that purpose each process is equipped with a counter being part of the process state which counts the number of messages the process has sent minus the number of messages it has received. (However, messages of the snapshot algorithm are not counted). By collecting and accumulating these counters together with the local snapshots the initiating process knows how many white messages have been in transit and can thus determine the end of the snapshot algorithm. (It knows how many copies it will get). Because after termination of the algorithm all processes are red and no white messages are in transit, a subsequent run can start without reinitialization simply by exchanging the roles of "white" and "red".

It is interesting to compare this algorithm to the snapshot algorithm by Lai and Yang [8] which also does not require FIFO channels. In their algorithm complete histories of sent and received messages are kept at every node. Since these message histories are considered to be part of the local state the initiator can compute the difference and thus determine the messages in transit without waiting for message copies. While this algorithm is "fast", it needs considerably more space than our scheme.

Finally, it should be noted that the snapshot algorithm can be used as a *distributed termination detection algorithm.* The local states are of no interest then, only the message counters are relevant. If the accumulated message counter equal zero then no messages cross the cut line and therefore the system is terminated [10].

Acknowledgments

The author would like to thank Michel Raynal and Gerard Tel for useful comments on a draft version of this paper.

References

[1] B. Awerbuch. Complexity of Network Synchronization. *Journal of the ACM*, 32(4), 1985. 804-823.

[2] G. Bracha and S.Toueg. Distributed Deadlock Detection. *Distributed Computing*, 2, 1987. 127-138.

[3] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1), February 1985. 63-75.

[4] C. J. Fidge. Partial orders for parallel debugging. In Barton Miller and Thomas LeBlanc, editors, *Proceeding of the ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging*, pages 183–194, University of Wisconsin, Madison, Wisconsin 53706, May 5-6 1988.

[5] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve Partial Ordering. In *Proceedings of 11th Australian Computer Science Conference*, pages 56–66, February 1988.

[6] Dieter Haban and Wolfgang Weigel. Global Events and Global Breakpoints in Distributed Systems. In *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, pages 166–175, January 1988.

[7] C. Jard and O. Drissi. Deriving Trace Checkers for Distributed Systems. Technical Report 347, IRISA, University of Rennes, France, 1987.

[8] Ten H. Lai and Tao H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25(3), May 1987. 153-158.

[9] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978. 558-565.

[10] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2, 1987. 161-175.

[11] Carroll Morgan. Global and Logical Time in Distributed Algorithms. *Information Processing Letters*, 20(4), May 1985. 189-194.

[12] G. Neiger and S. Toueg. Substituting for Real Time and Common Knowledge in Distributed Systems. Technical Report 86-790, Department of Computer Science, Cornell University, November 1986. Revised June 1987. Also see paper appeared in *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, 1987, 281-293.

[13] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Specifications and Domains. Part I. *Theoretical Computer Science*, 13, 1981. 85-108.

[14] P. Panangaden and K. Taylor. Concurrent Common Knowledge: A new Definition of Agreement for Asynchronous Systems. Technical Report TR 86-802, Department of Computer Science, Cornell University, 1986. Revised version TR 88-915, May 1988. Also in *Proceedings of ACM 5th Symposium on Principles of Distributed Computing*, 197-209, 1988.

[15] D. S. Parker, Jr. et al. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, 9(3), 1983. 240-247.

[16] Michel Raynal. A Distributed Algorithm to Prevent Mutual Drift Between n Logical Clocks. *Information Processing Letters*, 24, February 1987. Pages 199-202.

[17] J. F. A. K. Van Benthem. *The Logic of Time.* D. Reidel Publishing Company, 1983.

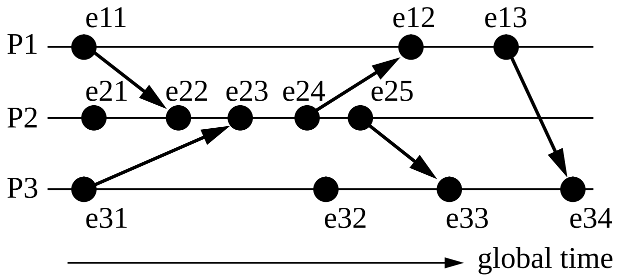

Fig. 1. An event diagram

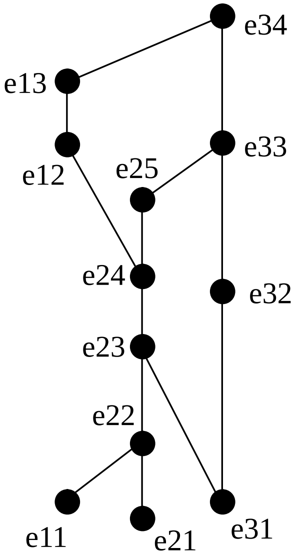

Fig. 2. The poset diagram

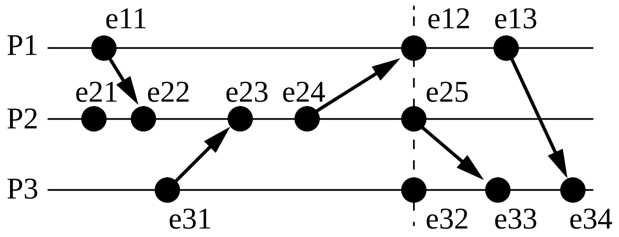

Fig. 3. An equivalent diagram

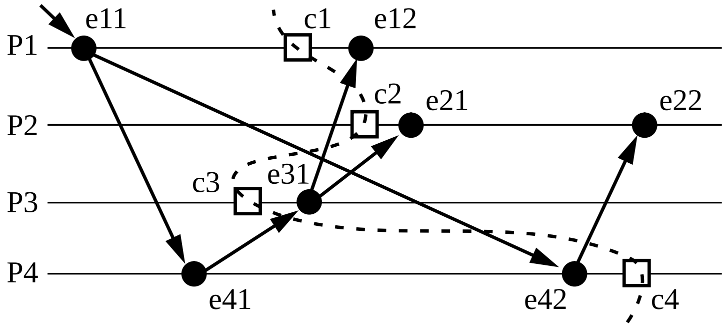

Fig. 4. A cut

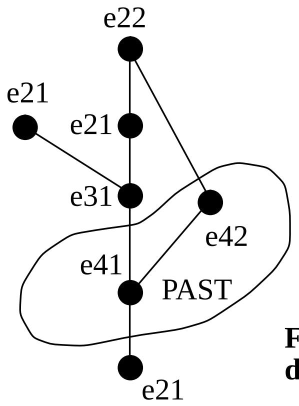

Fig. 5. The poset diagram of Fig. 4.

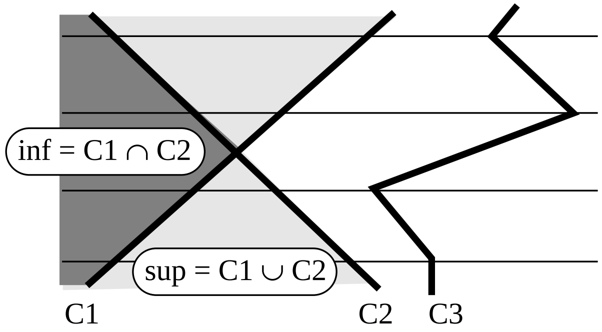

Fig. 6. The lattice structure of cuts

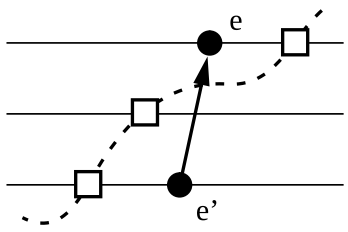

Fig. 7. An inconsistent cut

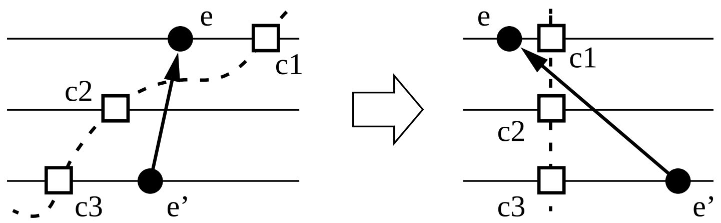

Fig. 8. The rubber band test

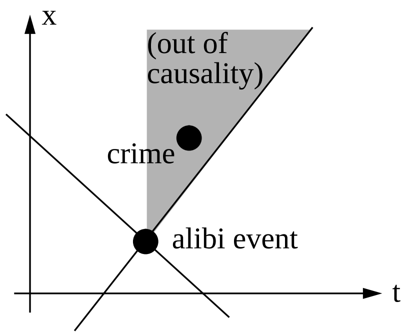

Fig. 9. Time and causality

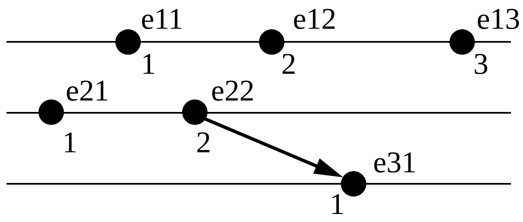

Fig. 10. Time-stamps of events

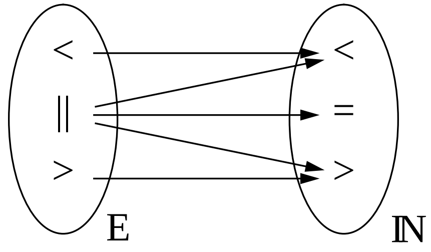

Fig. 11. Loss of structure

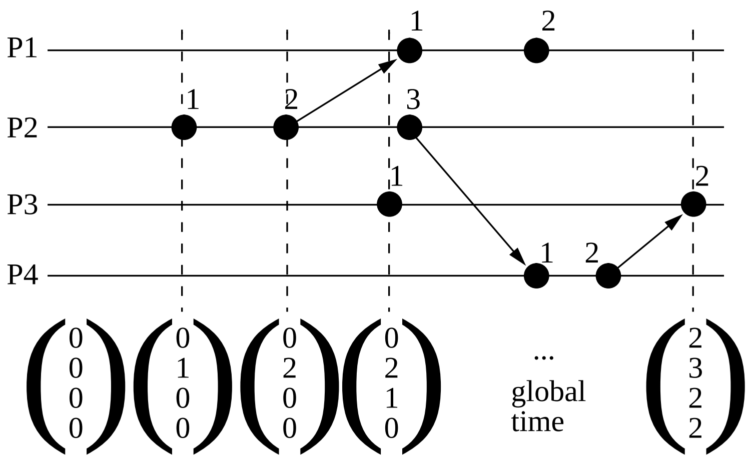

Fig. 12. Global vector time

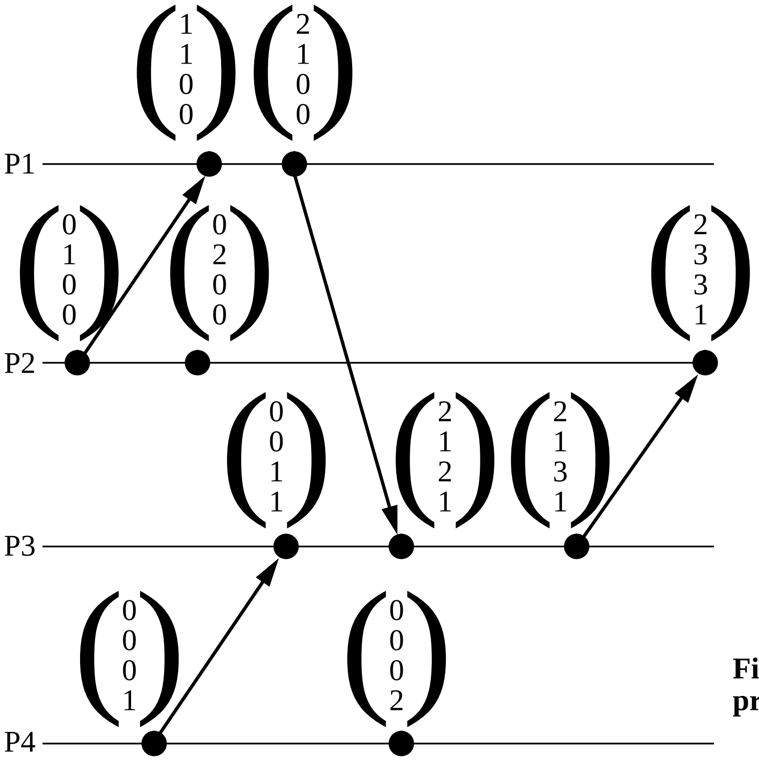

Fig. 13. Time propagation

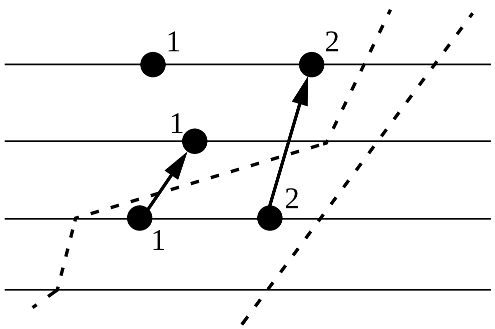

Fig. 14. Time of cuts

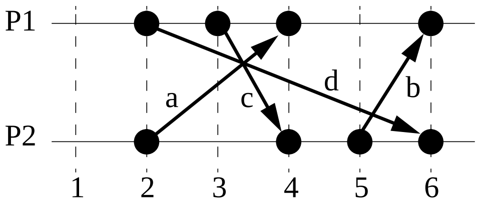

Fig. 15. A diagram with four messages

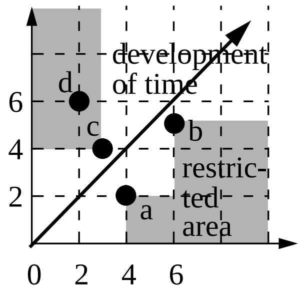

Fig. 16. Restrictions

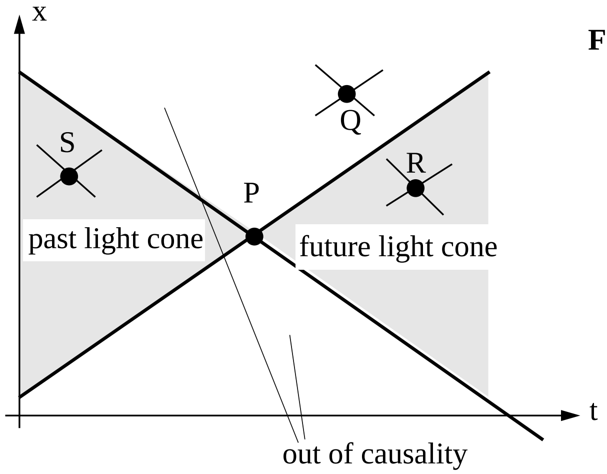

Fig. 17. Light cones

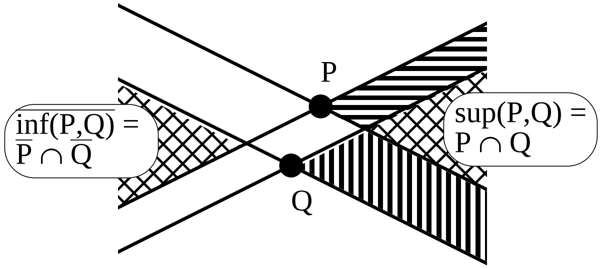

Fig. 18. Lattice structure of light cones