LOGICAL TIME

Friedemann Mattern

Darmstadt University of Technology

The notion of time seems to be elusive in distributed systems, where message delays are usually variable and where processes often do not have access to a global clock or to perfectly synchronized local clocks (*see* Clock Synchronization). On the other hand, 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 an alibi event and a criminal act. Hence, time is indeed useful, even in a distributed setting.

But what, in fact, is time? If one concentrates on the structural aspects, then the prevalent view of time is that of a set of “instants” with a temporal precedence order satisfying certain obvious conditions such as transitivity, irreflexivity, linearity, and density. Interestingly, however, in most cases when we make use of time and clocks we do not need all these properties – for example, digital clocks obviously do not realize the density axiom but are nevertheless useful in many cases.

The challenge consists in defining an abstract notion of time suitable for distributed systems which, on the one hand, is easily realizable without using physical clocks but, on the other hand, has enough interesting properties to justify the name “time”. Ideally, the concept of logical time should work as a partial substitute for real time and be practically useful in that respect. For example, one would like to be able to assign time values to events such that it is possible to infer potential causality between these events or to exclude causal influence in the sense that a “later” event cannot affect an “earlier” event.

Events, Causality, and Space-Time Diagrams

To model a distributed system, one typically considers processes which communicate by messages and which execute sequences of *events* (i.e., elementary or atomic actions). These events occur at specific instants in time. They are usually classified into *send* events, *receive* events, and *internal* events. An execution of a distributed system on such an abstract level can be depicted with the help of a *space-time diagram* (see the first figure) where time moves from left to right. Messages are drawn as arrows, and events are depicted by dots.

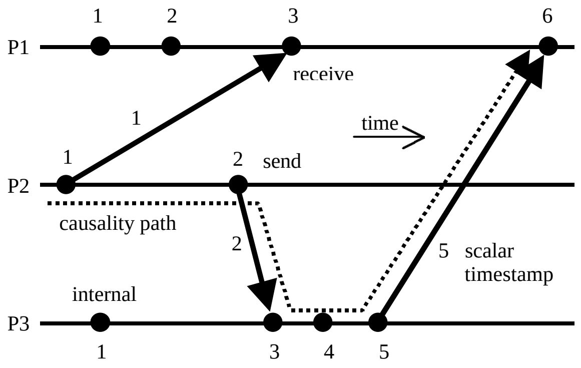
Events are *related* to each other: Events occurring at a particular process are linearly ordered by their local sequence of occurrence, and each receive event has a corresponding send event that happens earlier. Formally, one defines the *causality relation* ‘<’ as the smallest transitive relation on the set of events such that ‘<’ holds for any two events e, e’ if

1. e and e’ happen at the same process and e is the immediate predecessor of e’ or
2. e’ is the receipt of a message which was sent by event e.

It is reasonable to assume that in a space-time diagram message arrows do only move forward in time; therefore the causality relation ‘<‘ contains no cycles (i.e., it is a *partial order*). This relation is the heart of any sensible notion of logical time and determines its primary characteristic, namely that the *future cannot influence the past*. It was called “happened before” by Lamport [3], because e < e’ signifies that e happens before e’ – in the sense that e is drawn to the left of e’, (i.e., that e *happens earlier* than e’ in global time), but also in the sense that e *causally precedes* e’. The causality relation can be depicted on space-time diagrams by *paths*, which consist of message arrows and fragments of process lines traversed from left to right. Intuitively, this becomes evident if one reads e < e’ as “e may influence e’ ”. If two events e, e’ are *not* causally related (i.e., if they are *causally independent*) we write e || e’.

It is possible to view a space-time diagram as a *timing diagram* of an actual computation where the horizontal direction represents *real time*. Then the events occur at some specific instant of time as observed by an idealized external observer. It seems to be clear, however, that with respect to the causality relation the exact global time at which an event happens is of no concern, provided that the local sequence of events is not changed and message arrows do always point from left to right. Obviously, a space-time diagram can be transformed into another “equivalent” diagram (showing the same causality relation) by stretching and compressing the horizontal process lines as if they consist of idealized rubber bands. In that way, a set of mutually causally independent events may be aligned vertically as if they happen *simultaneously* – an important property for algorithms that compute consistent cuts and global states (*see* Distributed Snapshots).

Scalar Clocks



In an execution of a distributed system, where all actions are modeled by events, nothing happens between two successive events. Hence time needs only be advanced with the occurrence of an event and is therefore *discrete*. Since events are supposed to be durationless and occur at specific instants in time, a function $C: E \rightarrow T$ must be found which assigns a *timestamp* C(e) of a suitable "time domain" T to each event e of the event set E. The comparison of timestamps of different events should allow to draw certain conclusions about the relation of the events. It seems to be plausible that at least the notions "earlier" or "later" should exist within any sensible notion of time. Hence logical time should be a (possibly partial) *order* (T,<). From an abstract point of view, the function C can be called *logical clock*. A reasonable requirement on C is that it conforms to the causality relation:

$$\forall\ e, e' \in E: e < e' \rightarrow C(e) < C(e').$$

This important property is often called the *clock condition*. Stated verbally, it reads "an event e should get a smaller timestamp than an event e' if e can causally affect e' ". As a consequence of the clock condition the following properties hold:

1. For each process, time is *monotonically increasing*.
2. The logical time of a send event is always earlier than the logical time of the corresponding receive event.

In his seminal article "Time, Clocks, and the Ordering of Events in a Distributed System" [3], Lamport presented an implementation scheme for logical time based on an integer domain T for the timestamp values. It is realized by a system of counters C_i (one for each process) and a simple message handling protocol:

1. When executing an internal event or a send event at process P_i, clock C_i "ticks": $C_i := C_i + d$.
2. Each message contains a timestamp that 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: $C_i := \max(C_i, t) + d$.

Typically, d = 1, but any value d > 0 is acceptable. We define the *timestamp C(e) of an event* to be the value of the local clock just after it is being updated when executing the event. Observe that (if d = 1) this value is one larger than the length of the longest causality path that leads to the event.

As it stands, the scheme allows *different* events (in different processes) to have the *same* timestamp. For some applications (e.g., when the earliest request event should be granted access to an exclusive resource), timestamps must be *unique*. This is easily realized by using an ordering on process identifiers as a tie-breaking mechanism for such (causally independent) events.

Scalar clocks have some interesting applications. Unfortunately, however, they lose some structure by mapping the *partially* ordered events onto *linearly* ordered integers. In fact, events that are causally independent get assigned timestamp values as if they happen in a certain order. Hence, scalar clocks lack a desirable property of time: by checking the timestamps of events, it is usually not possible to assert that some event could *not* affect some other event. The reason for this defect is that C is an order-homomorphism which preserves '<' but which does not preserve negations (e.g., '||'). To repair this, the time domain T must represent the event structure in an *isomorphic* way, which means that the converse implication of the clock condition ($\forall$ e, e' $\in$ E: C(e) < C(e') $\rightarrow$ e < e') should also hold. This is in fact possible with a concept called "vector time".

Vector Clocks

To motivate vector time, assume that, similar to scalar clocks, each process P_i has a simple logical clock implemented by a counter, which is incremented by 1 each time an event happens. An idealized external observer who has 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 component for each process. The example depicted in the second figure illustrates the idea.

Because of variable message propagation delays, the instantaneous knowledge of the idealized observer cannot be realized in practice. Our more humble aim therefore consists of designing a mechanism by which each process can compute – without extra messages – an optimal *approximation* of this notion of global time. For this purpose each process should be informed at the earliest possible moment about all known events which did already happen.

To achieve this, we equip each process P_i with a clock C_i that consists of a vector of length n (where n is the total number of processes). Such a *vector clock* C_i is initialized with the null vector; it "ticks" immediately before the execution of an event by incrementing the value of its own component:

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

Each message is timestamped with the current value of the sender's vector clock. When receiving a timestamped message, a process combines its own time knowledge C_i with the timestamp t it receives by performing

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

where sup denotes the componentwise maximum operation. In this way, each process maintains knowledge about the number of events executed by all other processes of which it has heard. The *timestamp C(e) of an event e* occurring at process P_i is the value of clock C_i at the moment of the execution of e. (For receive events this is the value just *after* updating the clock.) The propagation of time knowledge and the updating of the vector clocks is shown in the figure.

Obviously, the events of process P_i are sequentially numbered by the i^{th} component of clock C_i. Or, to put it differently: before event e, C(e)[i]-1 other events did already happen on the same process. In fact, the vector timestamp C(e) of an event e contains in a compact way the complete knowledge about all those events from which e is (potentially) causally dependent. For example, C(e)[k]=j signifies that event e depends on the first, the second,... the j^{th} event of process P_k, but that it is not dependent of any later event of process P_k. Formally, the i^{th} component of C(e) can therefore be defined as follows:

$$C(e)[i] = |\{e' \mid e' \text{ is an event of process } P_i \wedge e' \leq e\}|.$$

(As usual, e $\leq$ e' stands for e < e' or e = e'.) One may easily check that this definition is realized by the previously mentioned rules for the handling of vector clocks and timestamps. In order to be able to compare time vectors, one defines the following relations for two vectors u, v:

$$u \leq v \;:\leftrightarrow\; \forall i\!: u[i] \leq v[i],$$

$$u < v \;:\leftrightarrow\; u \leq v \wedge u \neq v,$$

$$u \parallel v \;:\leftrightarrow\; \neg(u<v) \wedge \neg(v<u).$$

Observe that '$\leq$' (and hence also '<') is a *partial* order. Relation '$\parallel$', which is reflexive and symmetric (but non-transitive!), can be viewed as a generalization of the simultaneity of real time. However, whereas in real time simultaneity means truly identical instants in time, simultaneity in vector time has a larger extension.

The main property of vector clocks is that they induce an *isomorphism* of causal structure and temporal structure:

$$\forall e, e' \in E\!:\; e < e' \;\leftrightarrow\; C(e) < C(e').$$

This stronger form of the clock condition has an easy interpretation on space-time diagrams: An event e' has a larger timestamp than event e if and only if there is a path in the form of a *causal chain* from e to e'. Obviously, the value of a vector component can only increase along such a path. If, conversely, an event e' has a larger timestamp than another event e, then there must exist a path from e to e' along which the "time knowledge" C(e) is propagated. The reasoning with causal paths also reveals that the clock condition can be simplified if it is known that event e occurred at some process P_i and $e \neq e'$. Then only the vector components belonging to that process have to be considered: $e < e' \leftrightarrow C(e)[i] \leq C(e')[i]$.

The stronger form of the clock condition does immediately indicate how to test two events for causal independence:

$$\forall e, e' \in E:\ e \,||\, e' \ \leftrightarrow\ C(e) \,||\, C(e').$$

Informally, this asserts that exactly those events are mutually independent which happen simultaneously.

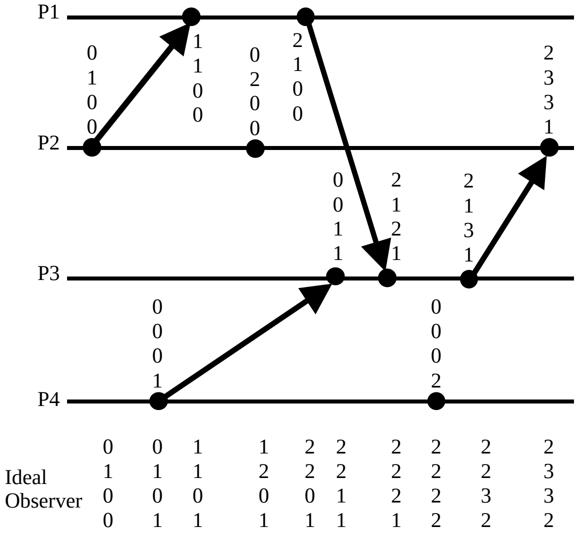
Applications of Logical Time

Logical time has many applications in the design of distributed algorithms. For example, finding the *earliest* event is useful for resolving conflicts (e.g., for guaranteeing mutual exclusion or for resolving deadlocks), and identifying a set of *simultaneous* events is helpful for determining consistent recovery lines.

Clearly, the causality relation is a powerful concept for analyzing and reasoning about distributed computations in general [4]. Closely related to causality is the notion of *consistency*, which is of utmost importance for the correct evaluation of *global predicates* (i.e., properties of the global state). Since vector clocks respect the stronger form of the clock condition and therefore fully characterize causality, they find many applications when causality or consistency must be handled in an operational way. For example, *consistent snapshots* are essentially subsets of events that are left-closed with respect to the causality relation and can therefore be determined using vector clocks.

Sometimes, it is necessary to *enforce causal order* among certain events. For example, an observer of a distributed system usually wants to receive notification messages from the different processes in “correct” event order so that a consistent global view may be obtained. If notification messages carry timestamps, an observer knows whether there are other notifications in transit referring to events that “happened before” – acceptance of messages that arrive “too early” can then be postponed.

Another application of logical time is *debugging* of distributed systems (*see* Distributed Debugging). With vector time it is possible to show that some event cannot be the cause for another event, thus helping to locate an error. Furthermore, timestamped trace data can be used to reduce the information necessary to *replay* a computation and to detect *race conditions*: a potential race condition exists if there is *no* causal relationship between two events. Since causally independent events may be executed concurrently, logical time is also useful to determine the *degree of parallelism* of a computation.



Besides scalar clocks and vector clocks, other logical clock systems were proposed in the literature. For example, to discard obsolete information in replicated databases, *matrix clocks* were introduced. Basically, they consist of n “parallel” vector clocks of length n: Clock M[j,k] of a process represents what the process currently knows about what process P_j knows about the number of events that process P_k already executed. Hence, “higher dimensional clocks” give processes additional (i.e., indirect) knowledge.

Scalar clocks are easily implementable. Vector clocks (and matrix clocks), on the other hand, are impractical or at least expensive – in particular for systems that consist of a large number of processes. Charron-Bost demonstrated [1] that in general the size of timestamp vectors cannot be reduced if the temporal structure should isomorphically represent the causal structure. There are several optimizations, however, which may substantially reduce the information in most practical situations [4]. For example, it is possible to piggyback on a message only the non-zero increments of those vector components that changed since the last communication with the same receiver and thus reduce the communication overhead.

More information on logical time may be found in [2]-[5], reference [5] is a collection of papers that also contains an annotated bibliography.

References

[1] B. Charron-Bost, *Concerning the Size of Logical Clocks in Distributed Systems*, Information Processing Letters 39, 11-16, 1991.

[2] C. Fidge, *Logical Time in Distributed Systems*, Computer 24, 28-33, 1991.

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

[4] R. Schwarz and F. Mattern, *Detecting Causal Relationships in Distributed Computations – in Search of the Holy Grail*, Distributed Computing 7, 149-174, 1994.

[5] Z. Yang and T. A. Marsland (eds.), *Global States and Time in Distributed Systems*, IEEE Computer Society Press, 1994.

Cross Reference:

Causality *see* Logical Time.

Clocks *see* Logical Time.

Consistency *see* Logical Time.

Events *see* Logical Time.

Global Predicates *see* Logical Time.

Happened Before *see* Logical Time.

Knowledge Propagation *see* Logical Time.

Lamport Clocks *see* Logical Time.

Logical Clocks *see* Logical Time.

Order *see* Logical Time.

Partial Order *see* Logical Time.

Snapshot *see* Logical Time.

Space-Time Diagram *see* Logical Time.

Time *see* Logical Time.

Timestamps *see* Logical Time.

Vector Clocks *see* Logical Time.

Vector Time *see* Logical Time.