Lecture 4
Distributed Time

(07-14,15-2020. Chapter 14)

Lecture Outline

```
1. Time
2. Clocks, events, and process states
3. Synchronizing physical clocks
4. Logical time and clocks
5. Global states
```

1. Time

(1) Time is important in DSs for many reasons.

a. Accounting/auditing.

b. Security and authentication (timestamps)

c. Maintenance of consistency of shared data.

(2) Consequences and conclusions drawn from Einstein's *Special Theory of Relativity*:

a. The speed of light is constant for all observers regardless of their relative velocity (still being studied and challenged).

b. Two events being judged to be simultaneous in one frame of reference are not necessarily simultaneous to observers in another frame of reference that are moving relative to it.

(a) An observer on the Earth and an observer on a spaceship flying away from Earth will disagree on the time interval between events.

(b) As speed of the spaceship increases, the disagreement also increases. If the spaceship could travel at the speed of time, the observer could *see/observe* events that were happening in the past (called *time channel*.

c. The relative order of two events can even be reversed for two different observers. Example: A traveler in a spaceship traveling at speed of light could observe events in the past that *are happening*, while to a static observer on the Earth, those past events had happened long time ago.

d. Non-existence of *absolute physical time*:
The classical notion of *absolute physical time* from Newton has been proven non-existing: there is no special physical clock in the universe to which we can appeal when we want to measure intervals of time.

(3) Difficulty in timestamping events in DSs.

a. As every where in the universe, there is no absolute, global time that we can appeal to.

b. Physical clocks at individual computers will drift and lack of absolute time makes it impossible to accurately synchronize them.

c. Even attempts of synchronizing physical clocks to be accurate to certain degree are difficult due to the need of message transfers and various of kinds of possible faults.

2. Clocks, events, and process states

(1) DSs: a DS consists of a set P of N processes $p_i, 1 \leq i \leq N$.

a. Each process executes on a single processor and processors do not share memory

b. At any given instance, a process p_i is in *local state* s_i. The process will transform to its next state s_{i+1} after the occurrence of the next *event*.

c. The local state of process p_i in p includes values of all the variables in it and values of any objects/resources in its local OS environment that p_i can access/modify.

d. An *action* that a process can take is either a communication action that sends or receives a message, or an internal action which affects its local environment.

e. An *event* is the occurrence of an action that a process takes.

(2) *Event ordering* within a single process and *history* of a process.

a. The sequence of events within a single process p_i can be placed in a single straight time line, which result in a *total ordering* (what is this?) denoted by $\rightarrow_i$:

(a) $e \rightarrow_i e'$ iff event e occurs before event e' in p_i.

(b) The assumption of a single processor makes this ordering well-defined, hence a total ordering (even in the presence of multi-threading of the OS).

b. The *history* of process p_i is a series of events that take place in it ordered by the total ordering relation $\rightarrow_i$:

$$history(p_i) = h_i =< {e_i}^0, {e_i}^1, {e_i}^2, \cdots >$$

(3) *Clocks*

a. Each event in a process is assigned a timestamp – a value recording the date and time of day of occurrence of the event.

(a) This timestamp can be either a hardware clock value $H_i(t)$ (generated through hardware interrupts) or a software clock value $C_i(t) = \alpha H_i(t) + \beta$, where parameters α and β are used to adjust software clock values (so that they can be more accurate).

(b) Since hardware clock cannot be completely accurate, the software clock values $C_i(t)$ will differ from the physical time t.

b. Successive events will correspond to different timestamps only if the *clock resolution* (the period between updates of the clock value) is smaller than the time interval between successive events.

c. *Clock skew* and *clock drift*, Fig.4.1, p.598

(a) *Clock skew*: the instantaneous difference between the readings of any two clocks is called their *skew*.

(b) *Clock drift*: physical clocks are usually crystal based. Manufactureship and interferences tend to make them operate at slightly different frequencies, termed as *clock drift*.

(c) *Clock drift rate*: is the change in the offset between the clock reading and a nominal perfect reference clock per unit of time measured by the reference clock. Usually in the range 10^{-6} or smaller.

(4) *Coordinated universal time* (UTC).

a. *International atomic time*: a standard second of IAT is defined as 9,192,631,770 periods of transitions between the two hyperfine levels of the ground state of Caesium-133 (Cs^{133}).

b. Why is UTC used as the acronym for Coordinated Universal Time instead of CUT?

The following is from (last verified available 06/04/2008)

http://www.boulder.nist.gov/timefreq/general/misc.htm

```
In 1970 the Coordinated Universal Time system was devised by an
international advisory group of technical experts within the
International Telecommunication Union (ITU). The ITU felt it
was best to designate a single abbreviation for  use in all
languages in order to minimize confusion.   Since unanimous
agreement could not be achieved on using either the English
word order, CUT, or the French word order, TUC, the acronym
UTC was chosen as a compromise.
```

c. *Astronomical time*: defined according to the rotation of the Earth on its axis and its rotation around the Sun.

d. The IAT and astronomical time (AT) have the tendency to get out of sync due to the instability of AT.

e. *Coordinated universal time* based on AT, but with the adjustment of inserting or deleting leap second to keep it synchronized with IAT.

(a) UTC signals are synchronized and broadcast globally and regularly from ground stations and satellites. GPS (global positioning system) is one source that broadcasts UTC.

(b) Signals received by individual computers with attached receivers have an accuracy in the order of 0.1 - 10 ms.

3. Synchronizing physical clocks

(1) The physical clock synchronization problem: how to synchronize your PC's clock? You may ask: synchronize with whom? This depends on your need:

a. Do you want to synchronize your clock as close as possible in reference to a standard clock source? Or

b. Do you want to synchronize your clock as close as possible with clocks in a group of other computers?

The former is called *external synchronization*, while the latter is called *internal synchronization*.

(2) *External synchronization* vs. *internal synchronization* (p.599). Assume I is a set of time intervals that will be taken by a UTC source. These two types of synchronizations are formally defined as follows.

a. *External synchronization*: For a synchronization bound $D > 0$ and for a source S of UTC time, $\forall t \in I | S(t) - C_i(t)| < D$, for $1 \leq i \leq N$.

b. *Internal synchronization*: For a synchronization bound $D > 0$ and for a source S of UTC time, $\forall t \in I | C_i(t) - C_j(t)| < D$, for $1 \leq i, j \leq N$.

c. Clocks that are externally synchronized are also internally synchronized and the reverse may not be true.

d. *Correctness* of clocks.

(a) A physical clock H is *correct* if its drift rate is within a known bound $\rho > 0$ (eg. $\rho = 10^{-6}$).

(b) Formally, a clock H is correct with respect to a drift rate ρ if for any two real time t and t':

$$(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$$

The correctness notion thus defined forbids bumps in the value of hardware clocks (hence software clocks too).

e. *Monotonicity* of clocks. A software clock C meets the *monotonicity* condition if it always advances:

$$t' > t \Rightarrow C(t') > C(t)$$

Clocks that do not satisfy the correctness condition can be adjusted to meet the monotonicity condition (to be seen shortly in Lamport's algorithm).

f. *Hybrid correctness*. A clock obeys the monotonicity condition and its drift is bounded *between synchronization points*, but is allowed to have bumpy drift ahead of synchronization points.

g. *Faulty* clocks.

(a) Clocks that are neither correct nor monotonic are called *faulty*.

(b) A clock that stops ticking completely is said suffering a *crash failure*.

(c) Any other clock failure is an *arbitrary failure*.

(3) Illustration of the problems with clock synchronization problem.

a. open an web page at the

URL http://www.time.gov/timezone.cgi?Central/d/-6/java

A clock showing the current CDT time will be displayed.

b. A clock skew value is also being displayed (such as *Accurate within 0.3 seconds*).

c. The JAVA program (actually a Java script) that displays the clock maintains a *virtual clock*.

* Observation 1: the Java script has to contact its external time source to re-synchronize with that source (this can be observed by disconnecting your PC from your ISP after displaying the URL). This involves *physical clock synchronization*.
* Observation 2: Displaying the URL page from different computers and different times may yield different *accuracy*. This indicates that accuracy is affected by communication *delays* between your browser and the external time server.
* Question: how does that Java script know the "accuracy" of current displayed clock?
* Answer: the Java script must be using certain algorithm to synchronize its clock with the external clock and to calculate "accuracy". The algorithm must take into accounts the comm. delays during synchronizations.

(4) Synchronization in a synchronous system

a. First attempt. Two processes are trying to synchronize their clocks internally.

(a) One process sends time t of its local clock to the other process.

(b) Assuming that it takes T_{trans} to deliver the synchronization message. The receiving process could set its clock value to $t + T_{trans}$.

b. Problem: the value of T_{trans} is difficult to obtain and estimate.

c. A minimum transmission time min can be estimated. This estimation can be done relatively accurately if the traffic between the two processes is very light.

d. In a synchronous system, there is an upper bound max on the time taken to deliver a message.

(a) Knowing the minimum possible transmission time min and maximum possible transmission time max, the *uncertainty* u in message transmission is

$$u = (max - min)$$

(b) If a receiving process sets its clock to be $t + min$ or $t + max$, then the clock skew is at most u in both cases. If set to $t + (max + min)/2$, then the clock skew is at most $u/2$.

(c) In general, in a system with N processes, the optimum bound on the clock skew is

$$u(1 - 1/N)$$

e. Most DSs are asynchronous.

(a) For such DSs, the value of max cannot be determined.

(b) For such DSs, the best we can say that $T_{trans} = min + x$ for some $x \geq 0$, which cannot be measured in theory.

(5) Cristian's synchronization algorithm (Cristian, 1989)

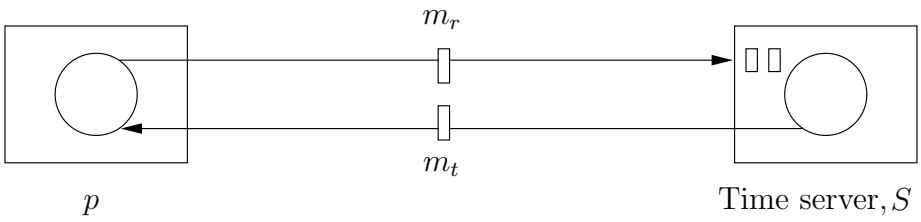

Figure 19: Clock synchronization using a time server (Cristian's algorithm)

a. Basic idea (Fig.14.2, p.601): the algorithm is a slight extension of the preliminary ideas of synchronization in synchronous systems. A time server connected to a device that receives UTC signals is configured. A client will send requests to the time server periodically which will send replies according to its clock values.

b. Basic observation and assumption:

   (a) Basic observation: even though there is no upper bound on message transmission delays in an asynchronous DS, the round-trip times between a pair of processes are often reasonably short, usually a fraction of a second.

   (b) Basic assumption: the observed round-trip times between a client and the time server are sufficiently short compared with the required accuracy.

c. The algorithm

   (a) A process p requests the time in a message m_r, and receives the time value t in a reply message m_t. To maximize accuracy, the server will insert the value t into m_t at the last possible instance before replying.

   (b) Process p records the total round-trip time T_{round} taken to send the request m_r and receive the reply m_t.

      · This round-trip time can be measured with reasonable accuracy if the clock drift rate is small.

      · Example: round-trip times on a typical LAN is in the order of 1-10 ms. A clock with drift rate of 10^{-6} (seconds/second) varies by at most 10^{-5} milliseconds.

   (c) The client process p will set its clock to $t + T_{round}/2$.

      · This calculation is quite accurate if the client and the server are on the same LAN segment.

      · If the client and server are on different networks, then the time taken to send m_r to server and the time taken to send reply m_t to client could be quite different. That would cause inaccuracy on the client clock.

d. Accuracy estimation. Assume that the minimum transmission time is known as min.

   (a) Observations:

      · The earliest instance at which S could have placed the time in m_t was min after p dispatched m_r;

      · The latest instance at which S could have placed the time in m_t was min before m_t arrives at p.

      · The width between the above two instances is $T_{round} - 2min$.

   (b) Based on above, the accuracy is

$$\pm(T_{round}/2 - min)$$

   (c) Accuracy can be improved by making several requests to S and taking the minimum (not average!) value of T_{round}.

(d) The greater is the accuracy required (hence the DS is busy and demanding), the smaller is probability of achieving it. This is so because the most accurate results are those in which both messages are transmitted in a time close to *min* – an unlikely event in a busy network.

e. Discussions

(a) Failure of the time server S may defeat the whole scheme. It was suggested that a group of time servers are used to improve availability of time servers. A client process can multicast its requests to all time servers and will only use the first reply received.

(b) A faulty time server that replied with spurious clock values, or an imposter time server that replied with deliberately incorrect times could corrupt the whole scheme. Again, a group of time servers can be used to resolve this problem. The issue is related to the distributed consensus problem to be covered in Chapter 12.

(6) The Berkeley algorithm (Gusella and Zatti, 1989)

a. Developed for systems of computers running Berkeley UNIX.

b. The algorithm:

(a) A single computer is designated as *master*. The rest of computers whose clocks are to be synchronized are called *slaves*.

(b) The master computer periodically *polls* slaves, which respond by sending back their own clock values.

(c) The master estimates their local clock times by observing the round-trip times (using techniques similar to in Cristian's algorithm). It averages values obtained (clock value plus half of round-trip time from each computer), including its own clock value.

(d) The master sends to each slave the amount by which the slave's clock has to be adjusted. This amount can be positive (the slave's clock is slower), or negative (the slave's clock is faster).

(e) Accuracy of the algorithm is improved by two special actions of the master:

· The master assumes a nominal maximum round-trip time between the master and slaves. With this, the master will eliminate any occasional readings larger than the nominal maximum round-trip time.

· The master also has a pre-defined specified amount by which a slave's clock can drift. The master takes so called *fault-tolerant average* – the average calculated from slave's responses that do not differ from one another by that specified amount. This eliminates the adverse effect of slaves' clocks that may shift significantly from time to time.

c. Experiments: on a system with 15 computers, maximum round-trip time taken to be 10 ms, the local clocks' drift rate was less than 2×10^{-5}, the algorithm synchronized the clocks within about 20-25 ms.

d. Failure of the master of course will defeat the whole algorithm. An election algorithm can be used to elect new master.

(7) The NTP (network time protocol)

a. NTP is an external syn algo intended to provide accurate time services over Internet. Main features and goals of NTP are:

(a) *Providing a service enabling clients across the Internet to be synchronized accurately to UTC*. NTP does so in the presence of variable and large delays of message transmissions. It employs statistical techniques to filter timing data and discriminates between the quality of timing data from different servers.

(b) *Providing a reliable service that can survive lengthy losses of connections*: Redundant servers and redundant paths between servers are configured, reconfiguration will be performed if necessary, to provide high available services.

(c) *Enabling clients to re-synchronize sufficiently frequently to offset rates of drifts found in most computers*: NTP can scale gracefully to serve a large number of clients and servers.

(d) *Providing protection against interference with the time service, whether malicious or accidental*: NTP clients and servers authenticate the sources of timing data. They also verify the addresses of messages received. (Hence messages from impostors can be detected).

b. Basic structures and ideas

(a) The servers in NTP are connected in a logical hierarchy. All servers connected in a logical hierarchy form a *synchronization subnet* (Fig.14.3, p.604). The levels of servers in a logical hierarchy is called *strata*.

(b) In a synchronization subnet:

· *Primary servers*, located at stratum 1, are directly connected to UTC sources.

· *Secondary servers*, located at stratum 2, receive timing data from (are synchronized with) primary serves.

· Similarly, servers at stratum 3 are synchronized with secondary servers, and so on.

· Those leaf servers execute at individual users' workstations.

Note: there are no pure clients in NTP. Secondary servers are the clients of primary servers; servers at stratum 3 are clients of secondary servers, and so on.

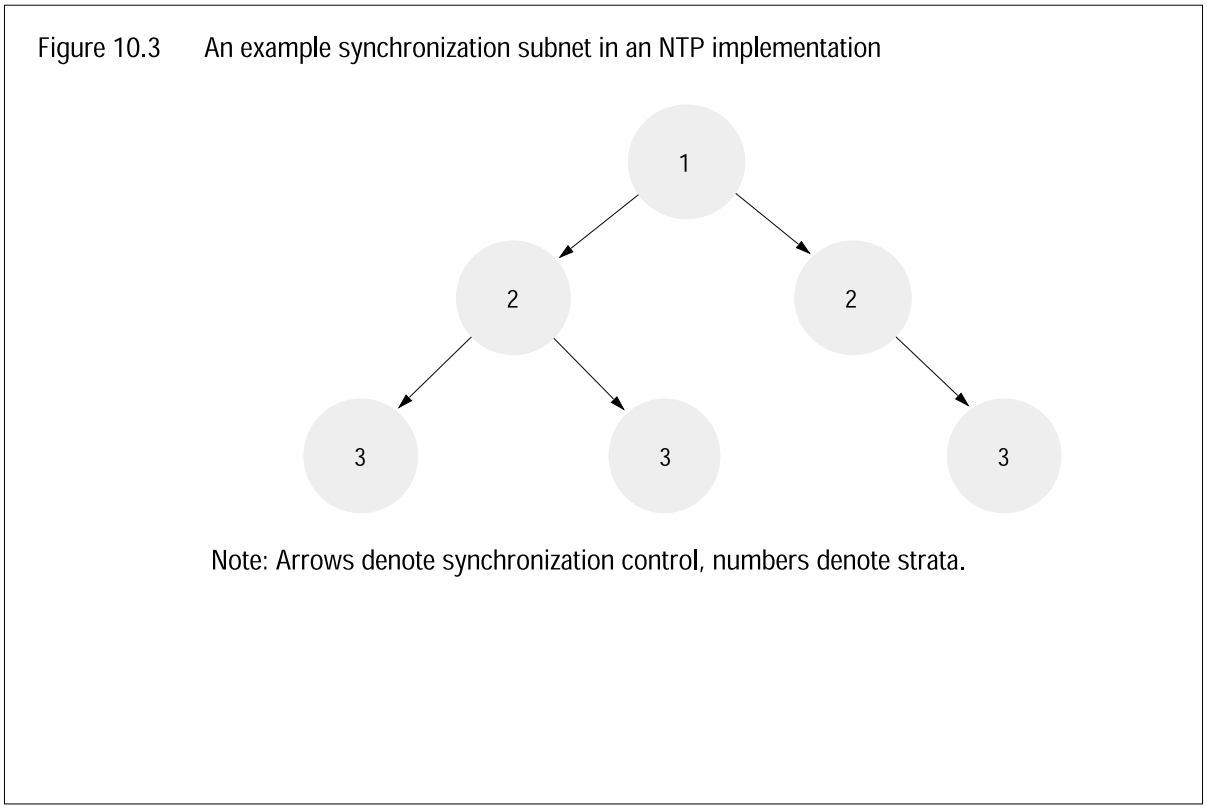

Figure 20: An example synchronization subnet in an NTP implementation (Fig.14.3,p.604)

(c) Due to the hierarchical structure, clocks in servers who are farther away to primary servers (namely who have a larger stratum numbers) are less accurate than those who are closer to the primary servers (errors accumulate at each level). NTP also uses the total round-trip delays to the root in accessing the quality of timekeeping data held by a particular server.

(d) A synchronization subnet can dynamically reconfigure as failures make timing sources unreachable or unavailable. Examples:

- If a lower level (with smaller stratum) server becomes unavailable, a connected higher level server (with larger stratum) may try to synchronize with another server (at a lower stratum or higher stratum).
- If the UTC source of a primary becomes unavailable, it may become a secondary server.

c. Operation modes: NTP servers synchronize with one another in one of three modes:

(a) *multicast* mode. Intended for use on high speed LANs, which have small delays and on which multicast/broadcast can be easily performed.

- One or more servers multicasts its clock value on a LAN, which will be

picked up by other servers attached to the LAN.

- This mode only achieves relatively low accuracies at a very low overhead.

(b) *procedure-call*.

- This mode is similar to Cristian's algorithm. One server accepts clock inquiries from other servers and replies with its current clock reading.
- This mode is used for applications where multicast mode cannot work, or higher accuracies are needed.

(c) *symmetric*.

- This mode can provide the highest accuracies. It can be used in multicast LANs or by higher levels of a synchronization subset to synchronize with one another (in order to achieve better accuracy. recall servers at higher levels have less accuracies).
- Under this mode, a pair of servers exchange timing data. This exchange can last a period of time and exchanged time data are retained in order to improve accuracy of their clocks.

UDP is the underlying transport protocol used by all three modes.

d. Details of *procedure-call* and symmetric modes.

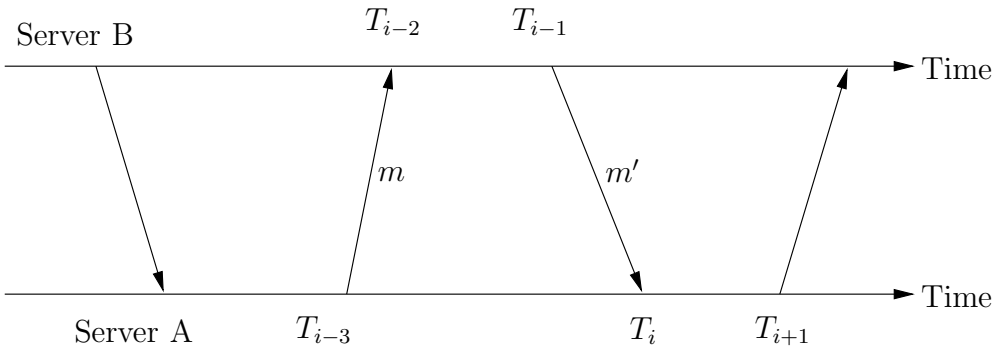

Figure 21: Messages exchanged between a pair of NTP peers (Fig.14.4,p.605)

(a) Processes (servers, or clients/servers) exchange pair of messages. Each such message bears three timestamps of recent message events (Fig.14.4, p.605). For example, when B sends m' to A, it includes T_{i-3}, T_{i-2}, and T_{i-1}.

- T_{i-3}: the local time when the previous message was sent;
- T_{i-2}: the local time when the previous message was received;
- T_{i-1}: the local time when the current message was sent;
- T_i: the local time when the current message was received;

Next time when A sends a message to B, it will include T_{i-1}, T_i, and T_{i+1}.

(b) Two more notes:

- For symmetric mode, the interval between the arrival of one message and the dispatch of the next can be non-negligible.
- Messages can be lost. After a loss, a re-transmitted message will bear a new T_{i-1}, but same T_{i-3} and T_{i-2}.

(c) For each pair of messages exchanged between two servers, the NTP calculates
- *an offset* o_i, which is an estimate of the actual offset between the two clocks; and
- *a delay* d_i, which is the total transmission time for the two messages.
- Let o be the true offset of the clock at B relative to that at A (remember we assume A starts the exchange).
- Let t and t' be the actual transmission times for messages m and m', respectively. Note: t and t' are *actually transmission times* for m and m' and cannot be actually measured.

(d) The following relationship holds:

$$T_{i-2} = T_{i-3} + t + o \quad \text{and} \quad T_i = T_{i-1} + t' - o$$

which can be better understood from:

$$t = T_{i-2} - T_{i-3} - o \quad \text{and} \quad t' = T_i - T_{i-1} + o$$

and be careful that:

$$t \neq T_{i-2} - T_{i-3} \quad \text{and} \quad t' \neq T_i - T_{i-1}$$

The above leads to:

$$o = T_{i-2} - T_{i-3} - t \quad \text{and} \quad o = T_{i-1} - T_i + t'$$

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

and we therefore have:

$$o = o_i + (t' - t)/2, \quad \text{where} \quad o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

Using the fact that $t, t' \geq 0$, and

$$t' = d_i - t \quad \text{and} \quad t = d_i - t'$$

$$o = o_i + ((d_i - t) - t)/2 \quad \text{and} \quad o = o_i + (t' - (d_i - t'))/2$$

i.e.

$$o = o_i + d_i/2 - t \quad \text{and} \quad o = o_i - d_i/2 + t'$$

we can have:

$$o_i - d_i/2 \leq o_i - d_i/2 + t' = o = o_i + d_i/2 - t \leq o_i + d_i/2$$

i.e.

$$o_i - d_i/2 \leq o \leq o_i + d_i/2$$

This shows that o_i is an estimate of o and d_i is a measure of the accuracy of this estimate. The following four examples demonstrate that o_i is a good estimate of o if t and t' stay close.

(e) Example 1: Assume $o = 0.1$ i.e. B's clock is 0.1 second faster than A's. Also assume that *actual time* t used to send m is the same as that of sending m' with value 0.5 seconds, i.e. $t = t' = 0.5$ seconds.

process A	process B	Notes
send(m) $T_{i-3} = 12.0$		At this point, B's clock has value 12.1
	receive(m) $T_{i-2} = 12.6$	At this point, A's clock has value 12.5
	send(m') $T_{i-1} = 13.2$	At this point, A's clock has value 13.1
receive(m') $T_i = 13.6$		At this point, B's clock has value 13.7

From above table, we have:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 = (12.6 - 12.0 + 13.2 - 13.6)/2 = 0.1$$

By the conclusion of NTP algorithm, it should have:

$$o = o_i + (t' - t)/2 = 0.1 + (0.5 - 0.5)/2 = 0.1$$

This shows that the estimated value is accurate and confirms the assumption that B's clock is 0.1 seconds faster than A's.

(f) Example 2: Here assume $o = -0.2$ (i.e. B's clock is 0.2 second slower than A's). Also that *actual time* t used to send m is the same as that of sending m' with value 0.5 seconds, i.e. $t = t' = 0.5$ seconds.

process A	process B	Notes
send(m) $T_{i-3} = 12.0$		At this point, B's clock has value 11.8
	receive(m) $T_{i-2} = 12.3$	At this point, A's clock has value 12.5
	send(m') $T_{i-1} = 13.4$	At this point, A's clock has value 13.6
receive(m') $T_i = 14.1$		At this point, B's clock has value 13.9

From above table, we have:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 = (12.3 - 12.0 + 13.4 - 14.1)/2 = -0.2$$

By the conclusion of NTP algorithm, it should have:

$$o = o_i + (t' - t)/2 = -0.2 + (0.5 - 0.5)/2 = -0.2$$

This again confirms the NTP's conclusion is correct.

(g) Example 3: Again assume $o = 0.1$ (i.e. B's clock is 0.1 second faster than A's). But we assume that *actual time* used to send m is 0.5 seconds and the *actual time* of sending m' is 0.7 seconds.

process A	process B	Notes
send(m) $T_{i-3} = 12.0$		At this point, B's clock has value 12.1
	receive(m) $T_{i-2} = 12.6$	At this point, A's clock has value 12.5
	send(m') $T_{i-1} = 13.2$	At this point, A's clock has value 13.1
receive(m') $T_i = 13.8$		At this point, B's clock has value 13.9

From above table, we have:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 = (12.6 - 12.0 + 13.2 - 13.8)/2 = 0$$

$$o = o_i + (t' - t)/2 = 0 + (0.7 - 0.5)/2 = 0.1$$

This again confirms that NTP's conclusion is correct.

(h) Example 4: Assume $o = -0.3$ (i.e. B's clock is -0.3 second faster than A's). But we assume that *actual time* used to send m is 0.9 seconds and the *actual time* of sending m' is 0.5 seconds.

process A	process B	Notes
send(m) $T_{i-3} = 12.0$		At this point, B's clock has value 11.7
	receive(m) $T_{i-2} = 12.6$	At this point, A's clock has value 12.9
	send(m') $T_{i-1} = 15.3$	At this point, A's clock has value 15.6
receive(m') $T_i = 16.1$		At this point, B's clock has value 15.8

From above table, we have:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 = (12.6 - 12.0 + 15.3 - 16.1)/2 = -0.1$$

$$o = o_i + (t' - t)/2 = -0.1 + (0.5 - 0.9)/2 = -0.3$$

(i) Summary of the four examples:

- NTP's conclusion is verified by the four different scenarios.
- The transmission delays of sending m and m' affect the value o_i obtained by A.

(j) NTP employs a data filtering algo to eight successive pairs $< o_i, d_i >$.

- The algo calculates the quality of estimate of o_i as a statistical quantity called *filter dispersion*.
- High filter dispersion means unreliable data.
- The value of o_j that corresponds to the minimum value d_j is chosen as o_i.

(k) Multiple sources may be used by a server to improve accuracy.

- NTP engages synchronization actions with several possible peers.
- Besides the filter algo, NTP also applies a peer-selection algo. Generally, peer servers that exhibit unreliable timing data as determined by the peer-selection algo will be eliminated.
- When the output of the peer-selection algorithm dictates change of a peer, NTP will try to select a peer that is at a lower stratum (they are closer to the primary server and hence are more accurate).
- In addition, peers with the lowest *synchronization dispersion* are favored. Syn dispersion is the sum of filter dispersions measured between the server and root of the synchronization subnet. Peers exchange synchronization dispersions in messages regularly.

e. A phase lock loop model is used to modify the local clock's update frequencies in terms of observations of its drift rate as determined from above discussion.

(a) This model reduces a clock's frequency if the clock is found faster and increases a clock's frequency otherwise.

(b) The model has a synchronization accuracy of one millisecond on LANs and an order of tens of milliseconds over Internet paths.

4. Logical time and clocks

(1) Introduction

a. Physical clock synchronization techniques are limited in accuracies. Therefore, in critical applications we cannot rely on physical time to find out the order of event executions in a DS.

b. In many applications, we are interested in the *relative order* of certain events, and we are not interested in the absolute physical timing of those events. Example: we want to make sure that a file is modified after a valid modification request has been issued.

(2) Lamport's method

a. The assumptions:

(a) Each individual process p_i is sequential, i.e. the events in a p_i are totally ordered according to their physical orders. This order is denoted as $\rightarrow_i$.

(b) The event of "sending" a message always occurs before the event of "receiving" the same message.

b. The *happened-before* relation: all events in a DS can be ordered with a simple scheme generalized from above assumptions. The resulted relation is called *happened-before* (or *causal ordering*, or *potential causal ordering*) relation, as defined formally:

HB1: if $\exists$ process $p_i : e \rightarrow_i e'$, then $e \rightarrow e'$.

HB2: For any message m, $send(m) \rightarrow receive(m)$, where $send(m)$ is the event of sending the message and $receive(m)$ is the event of receiving the same message.

HB3: Transitivity: if e, e', e'' are events and $e \rightarrow e'$ and $e' \rightarrow e''$ both hold, then $e \rightarrow e''$ also holds.

An example: Fig.14.5, p.607. Three processes, six events, and two message transmissions. We have $a \rightarrow f$, for instance.

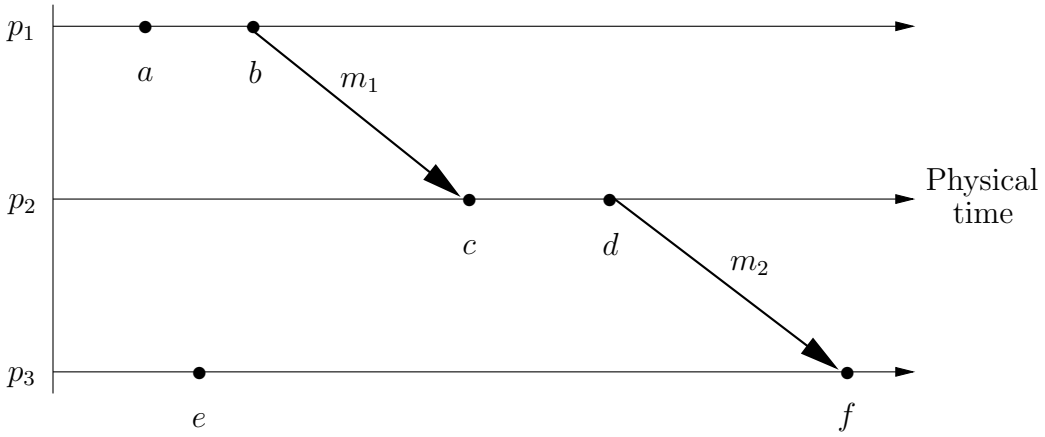

Figure 22: Events occurring at three processes (Fig.14.5,p.607)

c. Concurrent events and discussions

(a) Not all events in a DS are related by the happened-before relation $\rightarrow$.
- Example: Event a and e are not related, neither are c and e.
- Two events a and e that are not ordered by the relation $\rightarrow$ are said *concurrent* are denoted by $a \| e$.

(b) The relation $\rightarrow$ captures a flow of data based on sequential ordering of events within a process and on ordering of message passing. However, data can flow in ways other than message passing, which cannot be expressed by $\rightarrow$.

(c) If the relation $\rightarrow$ holds between two events, the first might or might not actually cause the second (A process could issue a new message after receiving a message. But that new message should be sent any way).

d. Logical clocks: introduced to capture and maintain the happened-before relation $\rightarrow$.

(a) Each process p_i maintains a monotonically increasing software variable L_i, called the logical clock for p_i. L_i is used to apply *Lamport timestamps* to events.

(b) Notation: for any event e at p_i, we use $L_i(e)$ to denote its timestamp. $L(e)$ is used to denote the timestamp of event e regardless process to which e belongs.

(c) The system of logical clocks $L_i, 1 \leq i \leq N$, are maintained according to the following rules:

LC1: L_i is incremented before each event is issued at p_i: $L_i = L_i + 1$

LC2: When a process p_i sends a message m, it attaches the value $t = L_i$ to m.

LC3: On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$, and then applies LC1 before timestamping the event $receive(m)$.

(d) Example and notes:
- Fig.14.6, p.609 shows the assignment of logical clock values to each event at three processes.
- The amount of increment at each process can be any positive value $\beta > 0$, not necessarily 1.
- By induction on the length of any sequences of events relating two events e and e', we can show that

$$e \rightarrow e' \Rightarrow L(e) < L(e')$$

(e) The reverse of above conclusion is not true: from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$.
Example: For the two events b and e in Fig.10.6 the relation $L(e) < L(b)$, but $b \| e$.

e. Totally ordered logical clocks

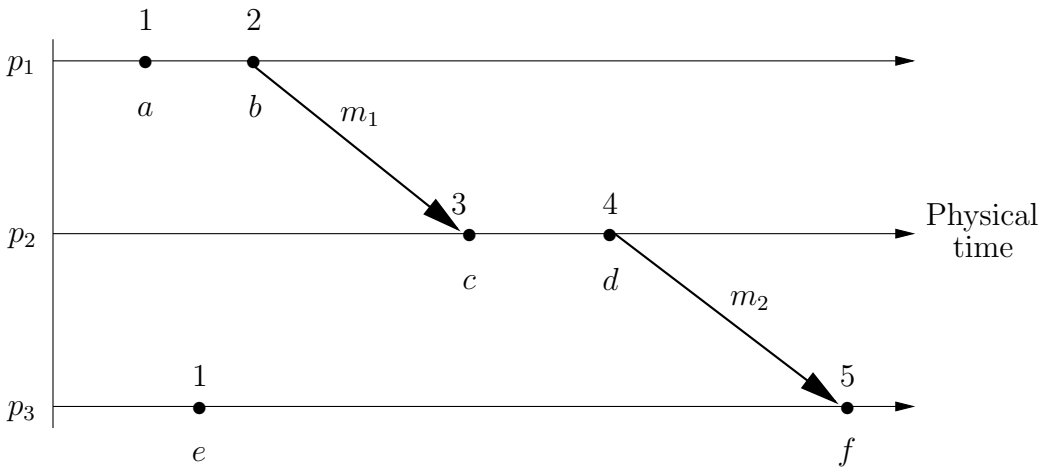

Figure 23: Lamport timestamps for the events shown in Fig.14.5 (Fig.14.6,p.609)

(a) It's possible for two events (from two different processes) to have same logical timestamp values.

(b) An artificial ordering of those pair of events can be created by including the identifier i of process p_i as part of the logical timestamps. Such timestamps are called *global logical timestamps*: (T_i, i). For two such timestamps, (T_i, i) and (T_j, j), $(T_i, i) < (T_j, j)$ if and only if either (i) $T_i < T_j$ or (ii) $T_i = T_j$ and $i < j$.

f. Vector clocks (Mattern [1989], Fidge[1991])

(a) Shortcoming of Lamport timestamps: from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$.

(b) The notion of *vector clocks* introduced can overcome this problem.

(c) Each process p_i now keeps a vector of N-component as a logical clock variable V_i. Each event e is associated with an N-component vector as its timestamp $V_i(e) = (i_1, i_2, \cdots, i_N)$:

VC1: Initially, $V_i[j] = 0$, for $1 \leq i, j \leq N$.

VC2: Just before p_i timestamps an event, it sets $V_i[i] = V_i[i] + 1$.

VC3: p_i includes the value $t = V_i$ in every message it sends.

VC4: When p_i receives a message with timestamp t in it, it sets $V_i[j] := max(V_i[j], t[j]), 1 \leq j \leq N$. This is termed as a *merge* operation (taking component wise maximum of two or more vectors).

(d) Example: Fig.14.7, p.610.

(e) Notes:

i. Two N-component vectors V and V' can be compared by comparing their corresponding components:

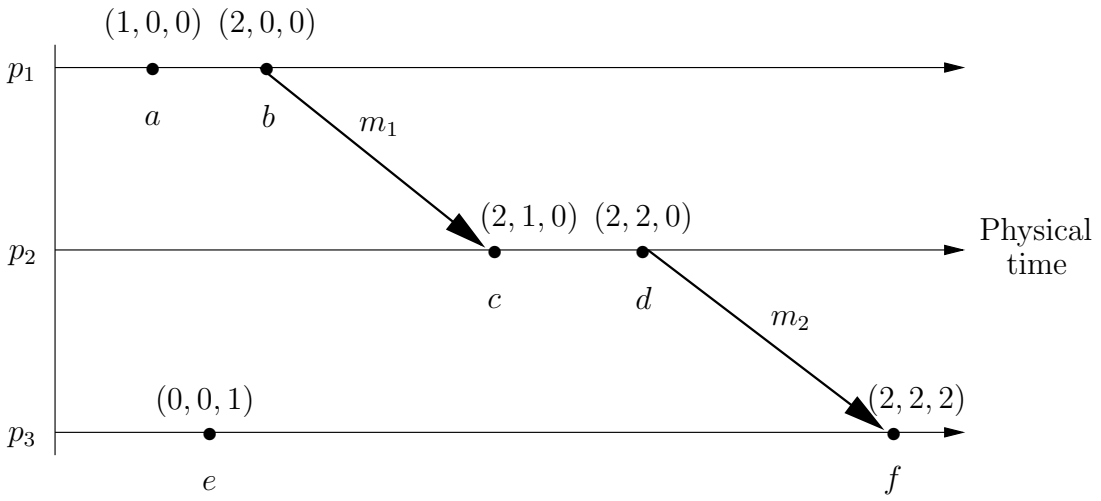

Figure 24: Vector timestamps for the events shown in Fig.14.5 (Fig.14.7,p.610)

$V = V'$ *iff* $V[j] = V'[j]$ for $j = 1, 2, \cdots, N$
$V \leq V'$ *iff* $V[j] \leq V'[j]$ for $j = 1, 2, \cdots, N$
$V < V'$ *iff* $V[j] \leq V'[j] \wedge V \neq V'$

ii. For any two events e and e', it is simple to see that

$$e \rightarrow e' \Rightarrow V(e) < V(e')$$

It is also can be shown that $V(e) < V(e')$ then $e \rightarrow e'$.

(f) Disadvantages of vector clocks: storage and message payload needed are proportional the size N of a DS.

g. Summary. Let P_1, P_2 and P_3 denote the following three properties:

$$P_1: \text{ if } e \rightarrow e' \text{ then } L(e) < L(e')$$

$$P_2: \text{ if } L(e) < L(e') \text{ then } e \rightarrow e'$$

$$P_3: \text{ if } e \neq e' \text{ then } L(e) \neq L(e')$$

For simplicity, only L is used in expressing P_1, P_2, and P_3. When discussing the vector timestamps, V should be used in place of L.

Method	P_1	P_2	P_3
Lamport's method without process ID	Always true	Not always true	Not always true
Lamport's total ordering method	Always true	Not always hold	Always true
Vector timestamps	Always true	Always ture	Always true

5. Global states

(1) *Global states*. Given a set P of N processes $p_i, 1 \leq i \leq N$ that represent a DS,

a. informally speaking a global state is a collection of states $s_1, s_2, \cdots, s_N$, where each of $s_i, 1 \leq i \leq N$ is a local state in process p_i that occurred at the some time instance t_i.

b. another form of global state definition requires that all states $s_1, s_2, \cdots, s_N$, to occur at the same time instance. There are two problems with that definition:

* It is difficult (impossible) to know that two events occur at the same physical time instance.
* It also introduces the so called *state explosion* problem.

(2) Applications of knowledge of global states. With the availability of global states many important problems in DSs can be easily solved (Fig.14.8, p.611):

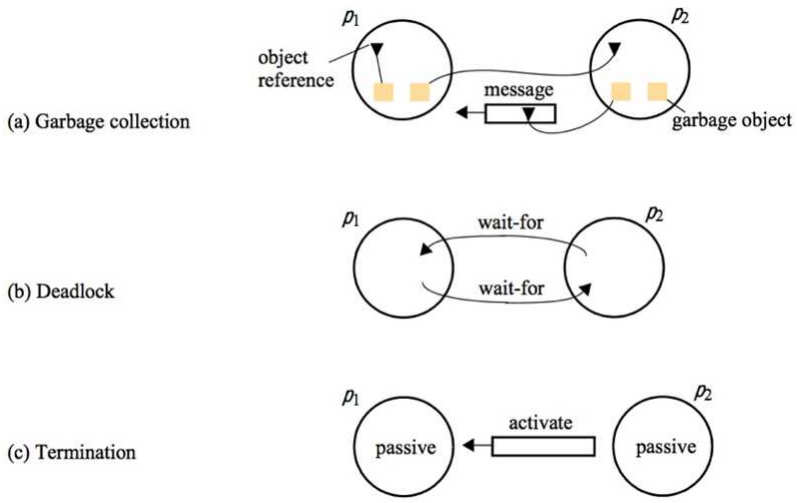

Figure 25: Detecting global properties (Fig.14.8, p.611)

a. *Distributed garbage collection*. A garbage is an object which is no longer being referenced by any processes in a DS. Hence the resources (such as memory) held by a garbage can be reclaimed by the system for reuse.

(a) Detecting whether an object is a garbage involves checking whether the object is still referenced by a process. However, in a DS an object can be referenced by a remote process (such as one of the two objects in p_1 in Fig.11.8).

(b) There is also the possibility that a process is sending a message to request access to an object which is not referenced at that given instance. Therefore the status of comm. channels should also be considered.

With global states available, we can just inspect global states from time to time to find and reclaim garbage.

b. *Distributed deadlock detection.*

   (a) The definition for deadlocks in a DS is the same as that in a centralized OS, except that in a DS a process may be holding a resource (local or remote) and waiting for allocation of another remote resource.

   (b) Therefore a distributed deadlock may involve processes and resources in a number of computers. Detecting distributed deadlocks demands cooperation of individual computers in a DS.

   Availability of global states will make deadlock detection trivial – we just inspect a global state to verify the existence of waiting cycles in it.

c. *Distributed termination detection.* This is a problem to detect whether a given distributed algorithm has terminates.

   (a) It involves verification of individual processes that collectively execute the distributed algorithm have terminated.

   (b) A process can be either in a passive, in which a process does not perform any activities, or active state. Because a passive process may be re-activated by a remote process through messages, detecting whether a process will no longer be active is not trivial.

   Again, if we can obtain global state information, then we can easily check whether an activation message is in transition at a given instance.

(3) Global states and consistent cuts

a. *History* (already defined at the beginning of this lecture): For a process p_i, the history of p_i is a complete sequence of events that occur in p_i:

$$history(p_i) = h_i =< {e_i}^0, {e_i}^1, {e_i}^2, \cdots >$$

b. *Finite prefix* h_i^k of history h_i:

$$h_i^k =< {e_i}^0, {e_i}^1, \cdots, {e_i}^k >$$

c. *Global history* of a DS:

   (a) For a process p_i, the notation s_i^k is used to denote the state of p_i immediately before the kth event occurs. Thus s_i^0 denotes the initial state of p_i, because it is the state that is immediately before the 1st event e_i^0.

   (b) The state of all open comm. channels should be considered as a factor in global state definition.

(c) A *global history* H of a given DS is the union of histories of individual processes in it:

$$H = h_1 \cup h_2 \cup \cdots \cup h_N$$

d. *Cuts* of histories :

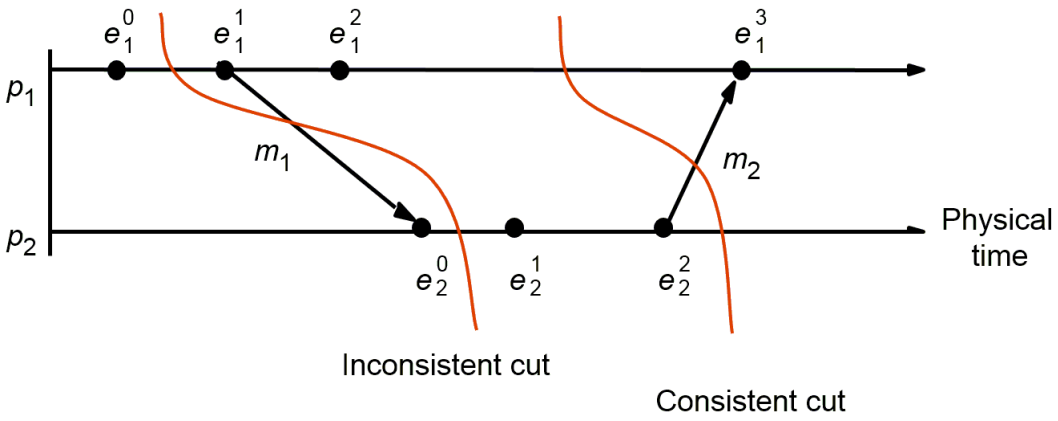

Figure 26: Cuts of a global state (Fig.14.9, p.613)

(a) A *cut* of a DS's execution is a subset of its global history. A cut must be a union of prefixes of process histories:

$$C = h_1^{C_1} \cup h_2^{C_2} \cup \cdots \cup h_N^{C_N}$$

(b) With definitions available so far, intuitively we would like to define a global state S to be a collection of states that correspond to the last states in a cut from each process.

The state s_i in the global state S corresponding to a cut C is a state from process p_i that is immediately after the last event $e_i^{C_i}$ performed by p_i in the cut, $1 \leq i \leq N$. The set of events $\{e_i^{C_i} : 1 \leq i \leq N\}$ is called the *frontier* of the cut.

(c) *Consistent cuts*. A cut C is *consistent* if for each event it contains it also contains all the events that *happened-before* that event:

$$\forall e \in C, f \rightarrow e \Rightarrow f \in C$$

Otherwise the cut C is said an *inconsistent cut*.

(d) *Consistent global states*: a consistent global state is one that corresponds to a consistent cut.
When we use the term *global states* we normally refer to consistent global states unless otherwise specified.

e. *Runs* and *consistent runs*:

(a) *Runs*. A *run* is a total ordering of all events in a global history H that is consistent with local history's ordering $\rightarrow_i$ of each process $p_i, 1 \leq i \leq N$.
A run is also called a *trace* in literature.

(b) *Consistent runs* (or *linearization*). A *consistent run* is a run that is consistent with the happened-before relation $\rightarrow$ on H.

(c) Notes: Consistent runs pass through only consistent global states. Not all runs pass through consistent global states.

(d) *Reachable states*. A state S' is reachable from a state S if there is a linearization that passes S and then S'.

(4) Global state predicate, stability, safety, and liveness

a. A global state predicate is a function that maps from the set of global states to the set {True, False}.

* Example 1: predicate α_1: the global state is not a deadlock state. For some global states the predicate α_1 is true. For some others α_1 is false. For these states we can apply certain deadlock resolution strategies to resolve the deadlock.
* Example 2: predicate α_2: there are some garbage objects in a global state. For some global states, the predicate α_2 is true. Then we can identify the garbage objects and reclaim allocated resources. For other global states α_2 is false, i.e. in these global states there are no garbage objects.

b. *Stability*: a global state is stable with respect to a given predicate α, if once the system enters a global state in which the predicate α is true, it will remain in all possible future global states reachable from that global state in which α will be true.

c. *Safety*. Given an initial global state S_0, an *undesirable* property β which is expressed as a predicate of global states (e.g. β could be the property of being deadlocked). *Safety* with respect to β and S_0 is the assertion that β will never evaluate to true for all states S reachable S_0.

Safety describes some undesirable properties that will never occur.

d. *Liveness*. Liveness is a property complementing the safety property. Given a *desirable property* α (such a specific distributed algorithm will eventually terminate) and an initial global state S_0, *liveness* with respect to α and S_0 is the property that, for any linearization L starting in the state S_0, α will evaluate to true for some state S_L reachable from S_0.

Liveness describes some desirable properties that are bound to occur, sooner or later.