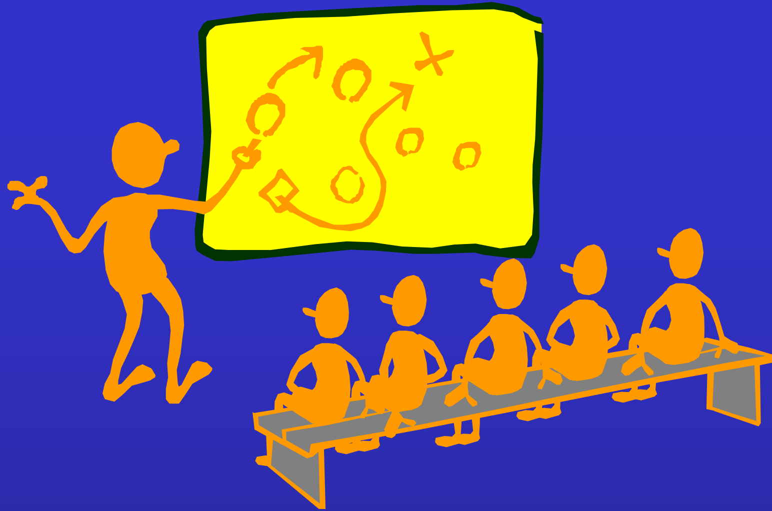


Distributed
operating systems

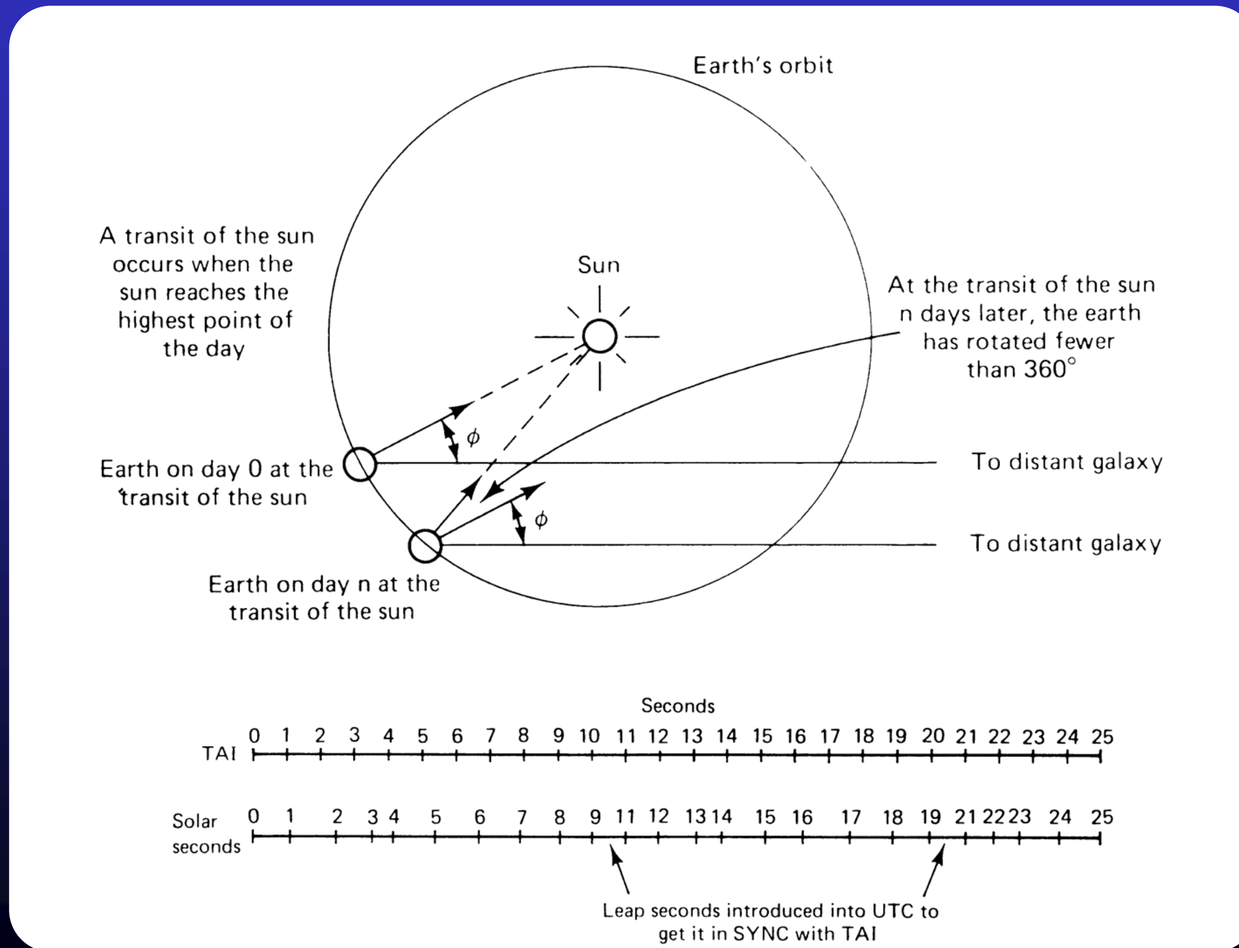
Synchronization in Distributed Systems

Clock synchronization

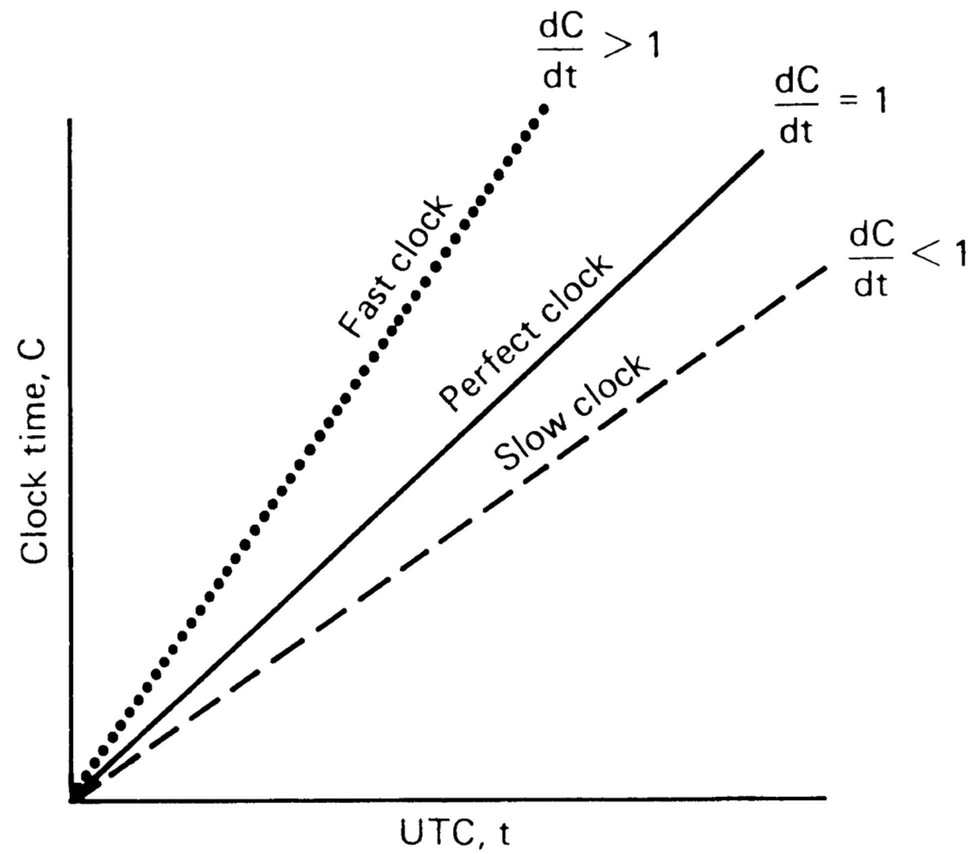
- Physical (real-time) clock synchronization
- Cristian's clock synchronization algorithm
- Berkeley clock synchronization algorithm
- Averaging clock synchronization algorithms (decentralized)
- Marzullo's algorithm
- Intersection algorithm
- Logical (Virtual-time) Clock Synchronization
  - Happened-before relation
  - Scalar logical clocks
  - Logical clock synchronization algorithm (Lamport)
  - Vector clocks (Fidge and Mattern)

Physical (real-time) clock synchronization

- Universal Coordinated Time (UTC) is the basis of all modern civil timekeeping. To provide UTC to people, who need precise time, the National Institute of Standard Time (NIST) operates a short-wave radio station with call letters WWV from Fort Collins, Colorado.



Clock synchronization



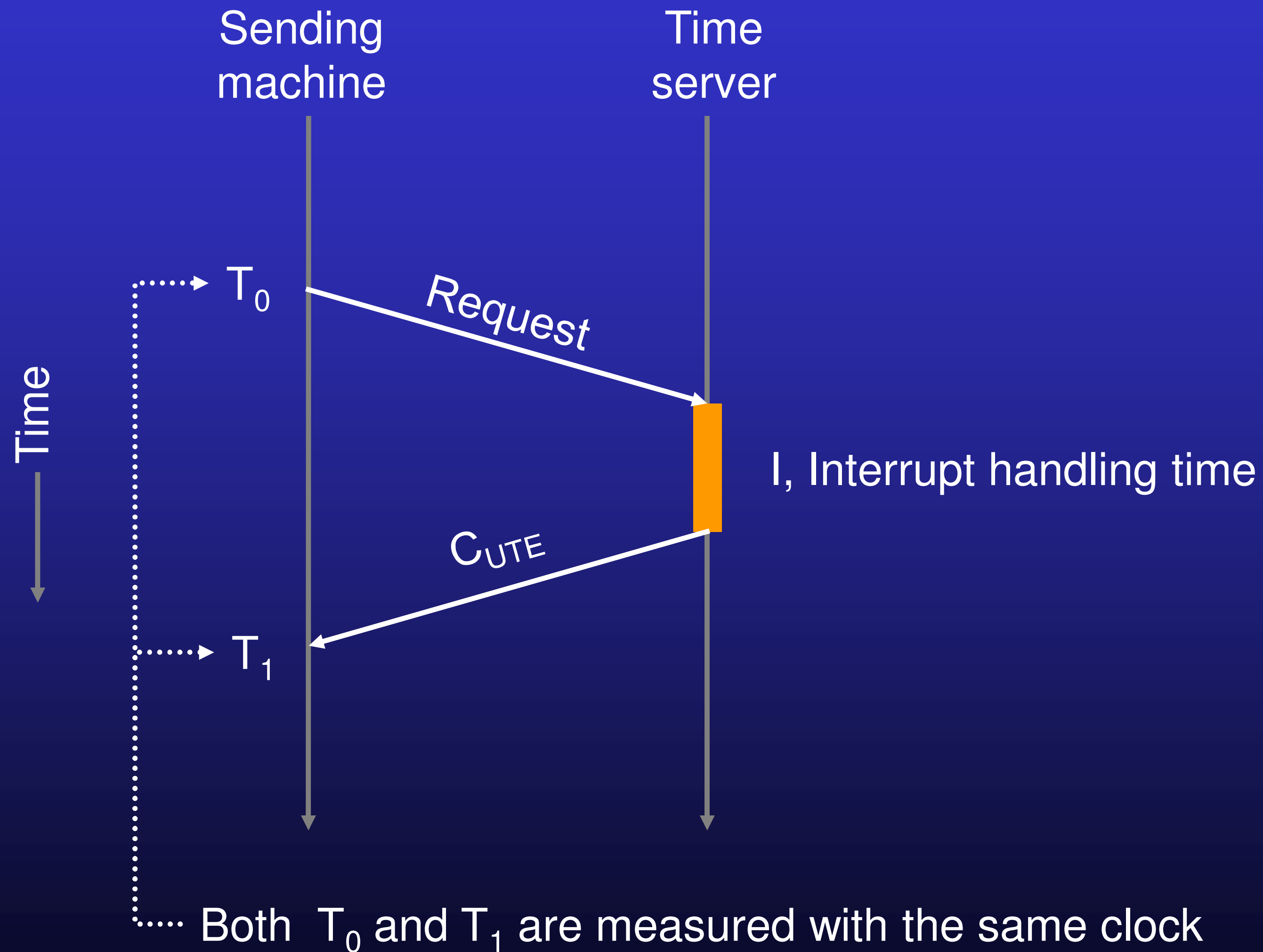
Cristian's clock synchronization algorithm[1]

This algorithm is well suited to systems in which one machine has a WWV receiver (it is called **time server**) and the goal is to have all the other machines stay synchronized with it.

- Periodically, each machine sends a message to the time server asking it for the current time. That machine responds as fast as it can with a message containing its current time, C_{UTC}.

- When the sender gets the reply it can:
  - just set its clock to C_{UTC},
  - increase the value in the message by propagation time,
  - additionally apply handle time.

Cristian's clock synchronization algorithm[2]



Berkeley clock synchronization algorithm

- The time server is active, polling every machine periodically to ask what time it is there. Based on answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved.

Averaging clock synchronization algorithms (decentralized)

- At the beginning of each interval, every machine broadcasts the current time according to its clock. After a machine broadcasts its time, it starts a local timer to collect all other broadcasts that arrive during some interval S. When all broadcasts arrive, an algorithm is run to compute a new time from them. The simplest algorithm is just to average the values from all other machines.

DTS and NTP

Two popular services for synchronizing clock and for providing timing information over a wide variety of interconnected networks are:

- the ***Distributed Time Service*** (DTS) and
- the ***Network Time Protocol*** (NTP).

b1

b1 Marzullo's algorithm
bartek; 2005-12-19

Marzullo's algorithm[1]

Marzullo's algorithm is used to estimate time on the basis of a number of noisy time sources. A modified version of this algorithm called the *intersection algorithm* is part of the Network Time Protocol.

The general idea of the algorithm is to take the smallest interval consistent with the largest number of sources, where *consistent intervals* are intervals which intersect.

Example:

- [7, 14], [13, 16], [10, 15]

  [13, 14] is consistent with all three intervals.
- [7, 8], [9, 13], [12, 15]

  There is no interval consistent with all three intervals but [12, 13] is consistent with the largest number of intervals.
- [1, 5], [4, 7], [6, 8]

  There is no interval consistent with all three intervals but there are two intervals consistent with the largest number of intervals: [4, 5] and [6, 7].

After determining an interval various approaches could be used to specify the result e.g.: the center of the interval or value calculated on the basis of a probabilistic model.

Marzullo's algorithm[2]

In the algorithm intervals [t-d, t+d] defining the time of sources are represented by two tuples of the form ⟨offset, type⟩: ⟨t-d; -1⟩ and ⟨t+d; +1⟩. In the tuples -1 stands for the beginning of a range and +1 represents the end.

Variables used in the algorithm:

best	the largest number of overlapping intervals found so far
current	a current number of overlapping intervals
[beststart, bestend]	the best interval found so far
i	an index
offset[i]	an offset of the ith *tuple*
type[i]	a type of the ith *tuple*

Marzullo's algorithm[3]

function ***getTimeInterval(*** *time intervals)*

1. **build** the list of tuples ⟨*offset; type*⟩

2. **sort** the list of tuples by the *offset*

/* If there are tuples with the same offset, one of the possible solutions is to put the tuples with the type +1 before the tuples of the type -1 to omit overlaps with no duration. */

3. *best* ← 0

4. *current* ← 0

5. **for each** tuple in the list in **ascending** order

6. *current* ← *current* – *type*[*i*]

7. **if** *current* > *best* **then**

8. *best* ← *current*

9. *beststart* ← *offset*[*i*]

10. *bestend* ← *offset*[*i* + *1*]

11. **return** [*beststart, bestend*]

Marzullo's algorithm[4]

Input: [3, 10], [1,6], [4, 8], [6,13], [9, 12]

Some steps from the algorithm:

[3, 10]	→	⟨3; -1⟩, ⟨10; +1⟩	ǀ
[1,6]	→	⟨1; -1⟩, ⟨6; +1⟩	ǀ
[4, 8]	→	⟨4; -1⟩, ⟨8; +1⟩	ǀ→→
[6,13]	→	⟨6; -1⟩, ⟨13; +1⟩	ǀ
[9, 12]	→	⟨9; -1⟩, ⟨12; +1⟩	ǀ

→→ ⟨1; -1⟩ ⟨3; -1⟩ ⟨4; -1⟩ ⟨6; +1⟩ ⟨6; -1⟩ ⟨8; +1⟩ ⟨9; -1⟩ ⟨10; +1⟩ ⟨12; +1⟩ ⟨13; +1⟩

Loops (lines 5.– 10.):

	tuple	current	best	[beststart, bestend]
1	⟨1; +1⟩	1	1	[1, 3]
2	⟨3; +1⟩	2	2	[3, 4]
3	⟨4; +1⟩	3	3	[4, 6]
…				
8	⟨10; -1⟩	2	3	[4, 6]
9	⟨12; -1⟩	1	3	[4, 6]
10	⟨13; -1⟩	0	3	**[4, 6]**

Intersection algorithm[1]

The Intersection algorithm is a modified version of Marzullo's algorithm. The goal of this algorithm is to produce the largest single intersection containing only truechimers. Truechimer is a clock that maintains timekeeping accuracy to a previously published (and trusted) standard, while a falseticker is a clock that does not *[RFC 1305]*. As in Marzullo's algorithm there are given m intervals of the form [t-d, t+d].

The main difference between these two algorithms is that the Marzullo's algorithm returns interval which does not necessarily include the center point of all the sources in the intersection. However, in the Intersection algorithm the result interval includes the interval returned by Murzallo's algorithm and the center point. In result such interval is larger and this allows to use some statistical data to select a point within the interval.

The Intersection algorithm looks for an interval with m-f sources, where f is the number of sources with the value outside the confidence band (wrong sources). To get the best result it is assumed that f should be as small as possible then the result is valid if $f < m/2$.

In comparison to Marzullo's algorithm for each interval there are three types of tuples:

1) the lower endpoint $\langle t-d; -1 \rangle$,
2) the midpoint $\langle t; 0 \rangle$,
3) the upper endpoint $\langle t+d; +1 \rangle$.

Intersection algorithm[2]

Variables used in the algorithm:

f	a number of false tickers
m	a number of all time sources
endcount	a number of the current endpoints
midcount	a number of the current midpoints
low	a value of the lower endpoint
high	a value of the upper endpoint
offset	an offset of the current *tuple*
type	a type of the current *tuple*

Intersection algorithm[3]

```
function getTimeInterval( time intervals )
1.  build the list of tuples ⟨offset; type⟩
2.  sort endpoint list by increasing offset || type
3.  for (f ← 0; f < m/2; f ← f + 1)             /* check if there are too many falsetickers */
4.       midcount ← 0
5.       endcount ← 0
6.       for each ⟨offset; type⟩ in ascending order    /* find lower endpoint */
7.            endcount ← endcount − type
8.            low ← offset
9.            if (endcount ≥ m − f) then break
10.           if (type = 0) then midcount ← midcount + 1
11.      endcount ← 0
12.      for each ⟨offset; type⟩ in descending order   /* find upper endpoint */
13.           endcount ← endcount + type
14.           high ← offset
15.           if (endcount ≥ m − f) then break
16.           if (type = 0) then midcount ← midcount + 1
17.      if (midcount ≤ f) then break        /* check whether too many midpoints are
              outside the found interval and continue until all falsetickers found */
18. if (low > high) then
19.      return FAILED             /* the proper intersection could not be found*/
20. return [low, high]
```

Intersection algorithm[4]

Input: [3, 10], [1,6], [4, 8], [6,13], [9, 12]

Some steps from the algorithm:

[3, 10]	→	⟨3; -1⟩, ⟨6,5; 0⟩, ⟨10; +1⟩	|
[1,6]	→	⟨1; -1⟩, ⟨3,5; 0⟩, ⟨6; +1⟩	|
[4, 8]	→	⟨4; -1⟩, ⟨6; 0⟩, ⟨8; +1⟩	|→→
[6,13]	→	⟨6; -1⟩, ⟨9,5; 0⟩, ⟨13; +1⟩	|
[9, 12]	→	⟨9; -1⟩, ⟨10,5; 0⟩, ⟨12; +1⟩	|

→→ ⟨1; -1⟩ ⟨3; -1⟩ ⟨3,5; 0⟩ ⟨4; -1⟩ ⟨6; -1⟩ ⟨6; 0⟩ ⟨6; +1⟩ ⟨6,5; 0⟩ ⟨8; +1⟩ ⟨9; -1⟩ ⟨9,5; 0⟩ ⟨10; +1⟩ ⟨10,5; 0⟩ ⟨12; +1⟩ ⟨13; +1⟩

Loops (lines 3.– 17.):

	f	m	[low, high]
1	0	10	[13, 1]
2	1	4	[6, 6]
3	2	2	**[4, 10]**

Happened-before relation[1]

- One can point out that clock synchronization need not to be absolute. If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems. For a certain class of algorithms, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time.

- The ***happened before relation (causal precedence relation)***, denoted by $\rightarrow$, on a set of events satisfies the following conditions:

1) If a and b are events in the same process and a occurs before b, then $a \rightarrow b$.
2) If a is the event of sending a message by one process and b is the event of the receipt of the same message by another process, then $a \rightarrow b$.
3) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. That is, happen before is a transitive relation.

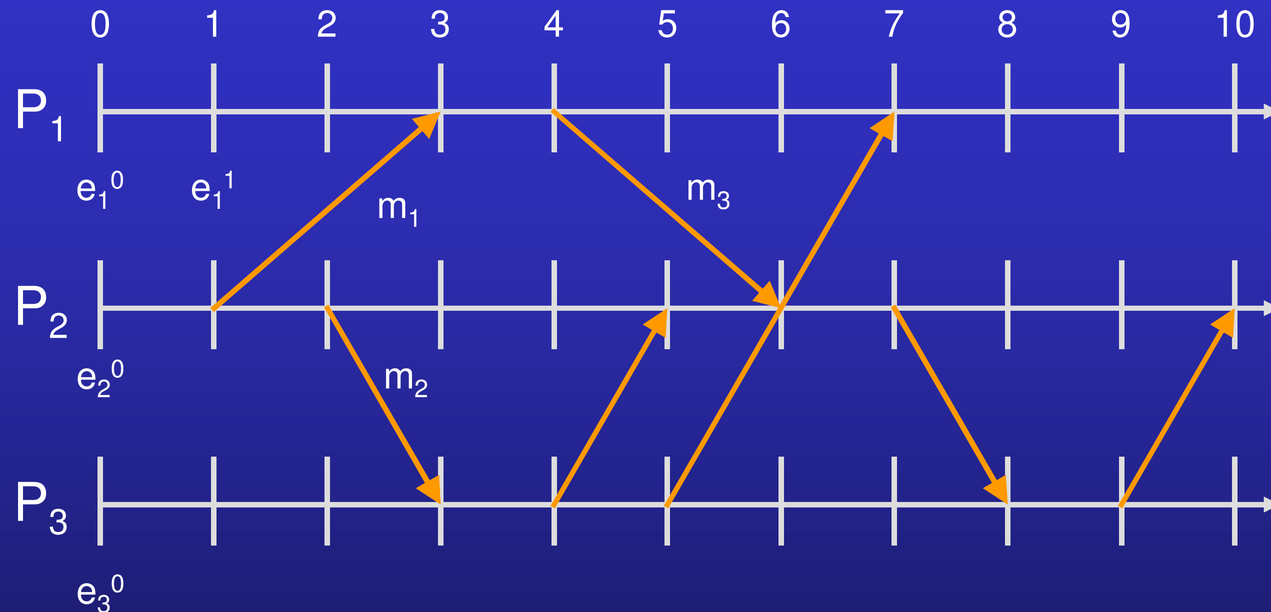
Happened-before relation[2]

Formally:

$e_i^k \rightarrow e_j^l \Leftrightarrow$

1) $i=j \wedge k<l$,
2) $i \neq j$, a e_i^k is the event of sending a message m by process P_i and e_j^l is the event of the receipt of the same message m by process P_j,
3) there exists a sequence of events $e_0\ e_1 \ldots e_n$, such that $e_0 = e_i^k$, $e^n = e_j^l$ and for each pair (e^x , e^{x+1}), where $0 \leq x \leq n$ point 1 or 2 holds.

Happened-before relation – space-time diagram



If $e_i^k \rightarrow e_j^l$ or $e_j^l \rightarrow e_i^k$ then these events are called ***causally dependent***, otherwise ***causally independent*** or ***concurrent***. Concurrent events e_i^k and e_j^l are denoted by $e_i^k \parallel e_j^l$.

Scalar logical clocks

Logical clock is a function (mechanism) assigning to each event *e* a value *C(e)* (timestamp) satisfying the following condition:

$$e_i^k \rightarrow e_j^l \;\Rightarrow C(e_i^k) < C(e_j^l)$$

Logical clock synchronization algorithm (Lamport)[1]

- Each process P_i increments C_i by d between any two successive events.

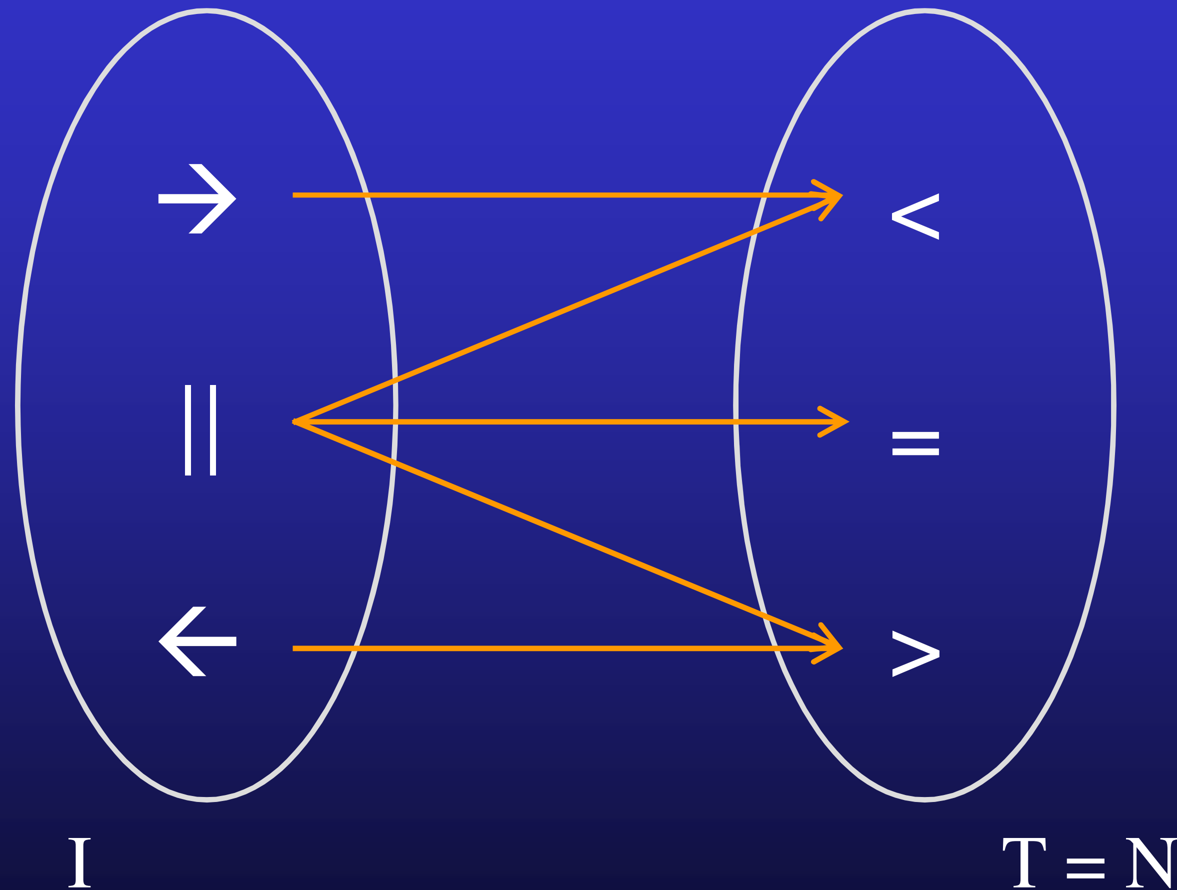
- If a is the event of sending a message m by process P_i, then message m contains a timestamp $T_m = C_i(a)$ and upon receiving the message m a process P_j sets:
  - $C_j = max(C_j , T_m) + d$ or
  - $Cj = max(Cj , T_m + d)$

- Note, that Lamport's clock satisfies condition:

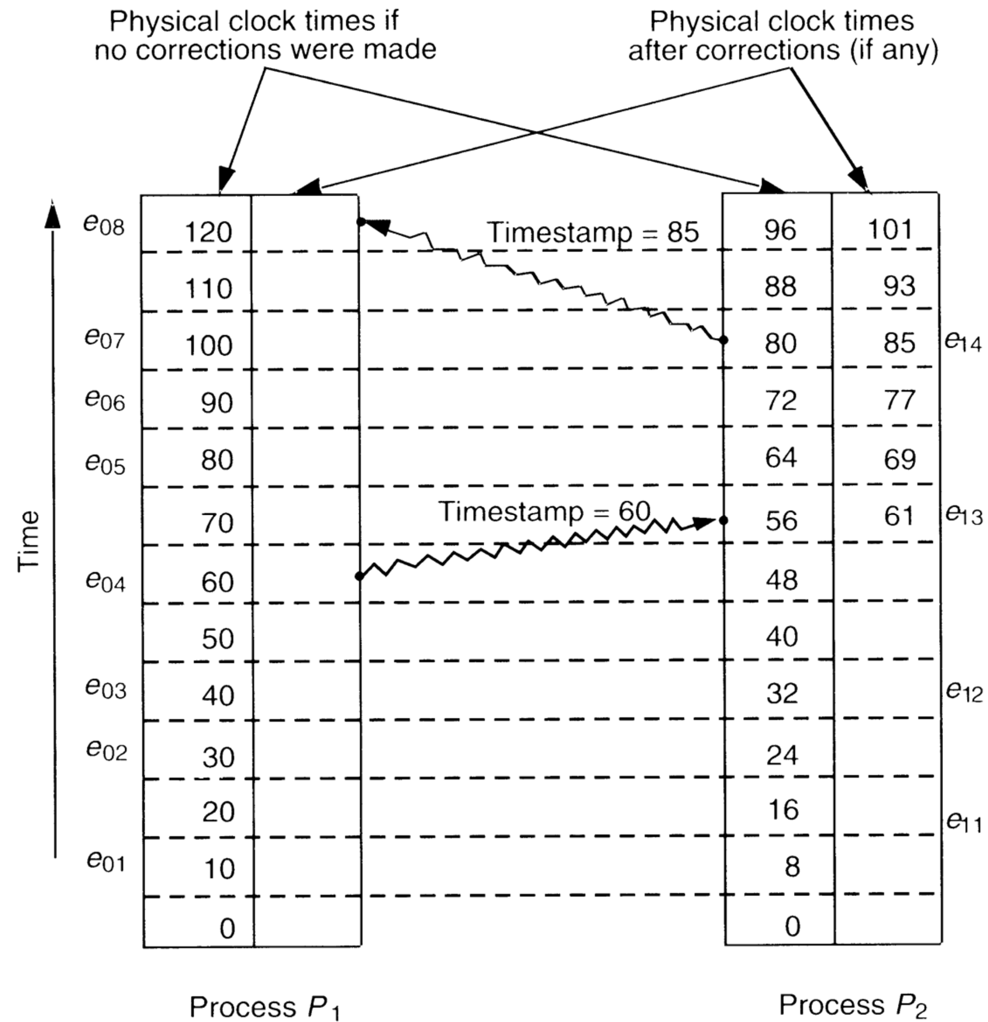
$$e_i^k \rightarrow e_j^l \Rightarrow C(e_i^k) < C(e_j^l)$$

but the reverse implication is not true!!

Logical clock synchronization algorithm (Lamport)[2]



Implementation of logical clocks by using physical clocks



Vector clocks (Fidge and Mattern)

Each process is equipped with a clock C_i, which is an integer vector of length n. The clock C_i can be thought of as a function that assigns a vector $C_i(a)$ to any event a. $C_i(a)$ is referred to as the timestamp of event a at P_i. $C_i[i]$ the ith entry of C_i corresponds to P_i's own logical time. $C_i[j]$, for $j \neq i$ is P_i's best guess of the logical time at P_j. More specifically, at any point in time, the jth entry of C_i indicates the time of occurrence of the last event at P_j which "happened before" the current point in time at P_i.

Vector clock implementation

- Clock C_i is incremented between any two successive events in process. P_i: $C_i[i] := C_i[i] + d$.

- If event a is the sending of the message m by process P_i, then message m is assigned a vector timestamp $T_m = C_i[n]$.

- On receiving the same message m by process P_j, C_j is updated as follows:

$$\forall k\ C_j[k] := max(\ C_j[k], T_m[k]\ )$$

- Note, that on the receipt of message, a process learns about the more recent clock value of the rest of processes in the system.

Vector clocks

Theorem. At any instant:

$$\forall i,j \;\; C_i[i] \geq C_j[i]$$

Theorem. In the system of vector clocks:

$$e_i^k \rightarrow e_j^l \;\Leftrightarrow\; C(e_i^k) < C(e_j^l)$$

Mutual exclusion

- A Centralized Approach
- Distributed Algorithms
  - Lamport's algorithm
  - Ricart-Agrawala algorithm
  - Maekawa's algorithm
- Token-Based Algorithms
  - Suzuki-Kasami's broadcast algorithm
  - Raymond's tree-based algorithm

A centralized approach[1]

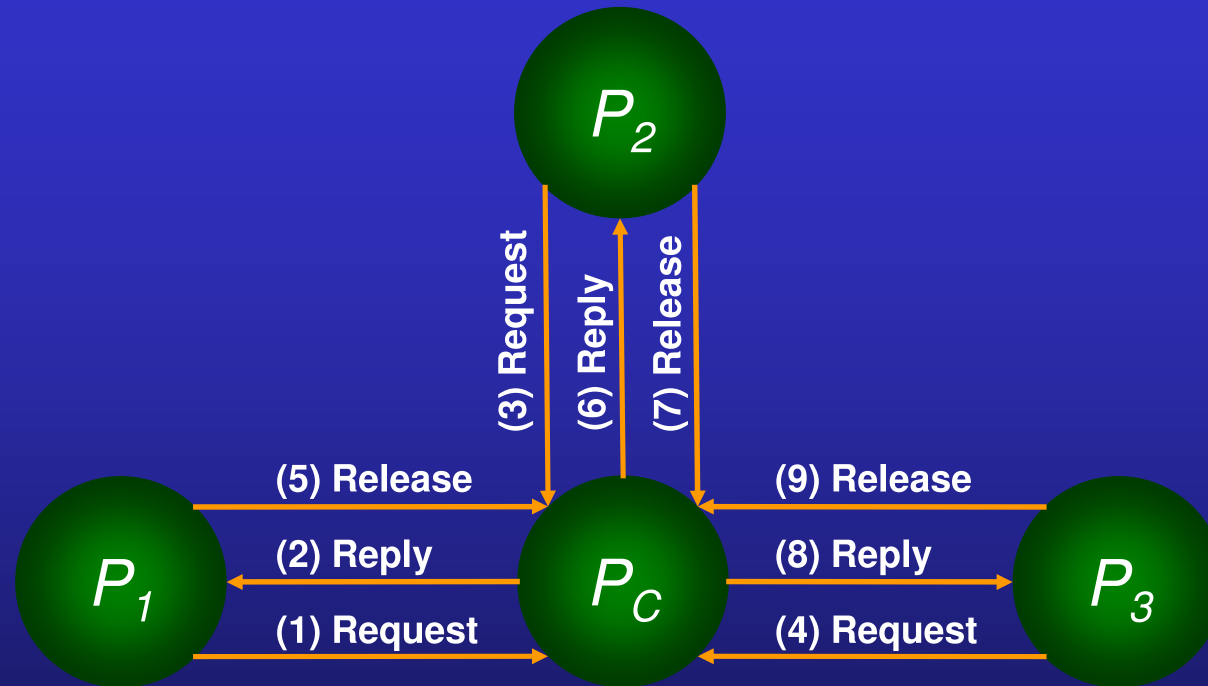
1) One process is elected as the coordinator.
2) Whenever process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission.
3) If no other process is currently in that critical region, the coordinator sends back a reply granting permission. When the reply arrives, the requesting process enters the critical region.
4) When another process asks for permission to enter the same critical region, the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied".

A centralized approach[2]

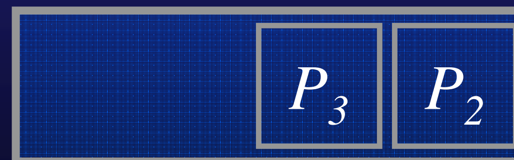
Properties of the algorithm:

- it is fair,
- no starvation,
- easy to implement,
- requires only three messages,
- the coordinator is a single point of failure.

A centralized approach[3]



Status of
request queue



Lamport's algorithm[1]

Lamport was the first to give a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme. Let $\mathbf{R_i}$ be the ***request set*** of site S_i, i.e. the set of sites from which S_i needs permission when it wants to enter CS. In Lamport's algorithm, $\forall i : 1 \leq i \leq N : \mathbf{R_i} = \{S_1, S_2, \ldots, S_N\}$. Every site S_i keeps a queue, ***request_queue***$_i$, which contains mutual exclusion requests ordered by their timestamps. This algorithm requires messages to be delivered in the FIFO order between every pair of sites.

Lamport's algorithm[2]

Requesting the critical section.

1) When a site S_i wants to enter the CS, it sends REQUEST(ts_i, i) message to all the sites in its request set $\mathbf{R_i}$ and places the request on $request_queue_i$ (ts_i is the timestamp of the request).
2) When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it returns a timestamped REPLY message to S_i and places site S'_i *s* request on r$equest_queue_j$.

Executing the critical section.

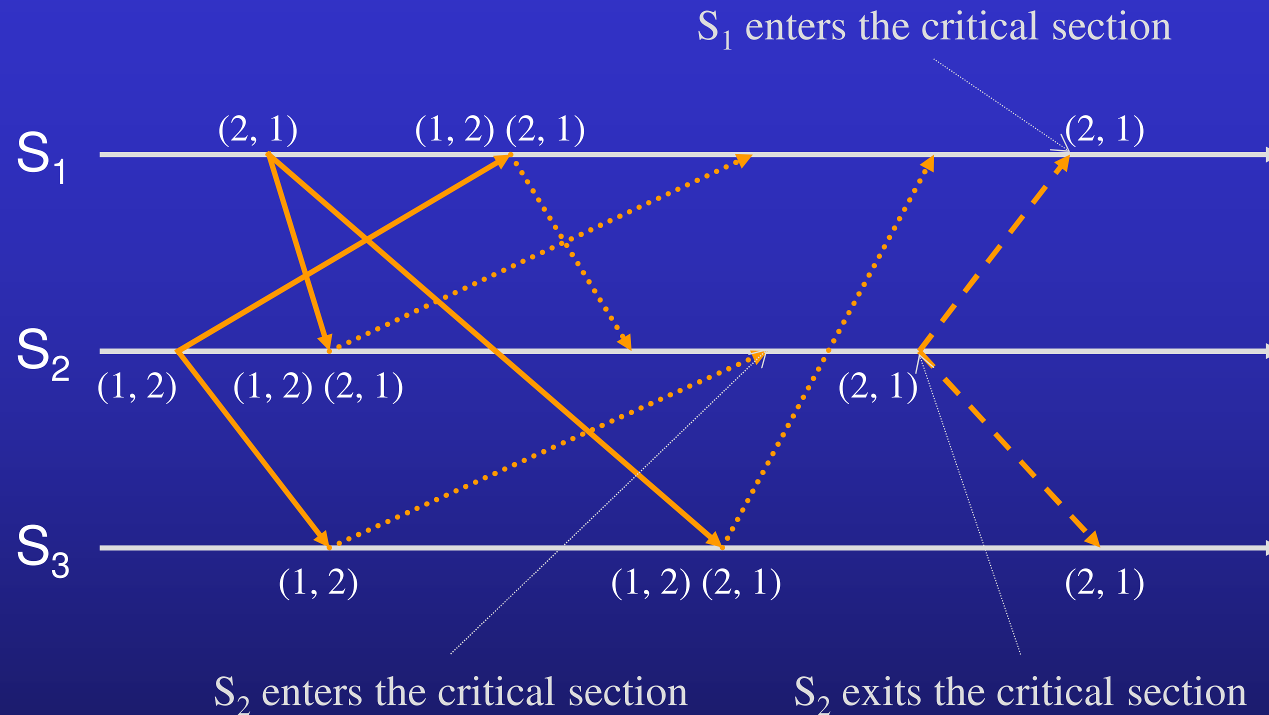
1) Site S_i enters the CS when the two following conditions hold**:**
   a) [L1:] S_i has received a message with timestamp larger than (ts_i, i) from all other sites.
   b) [L2:] S'_i*s* request is at the top $request_queue_i$.

Releasing the critical section.

1) Site S_i, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.
2) When a site S_j receives a RELEASE message from site S_i, it removes S'_is request from its request queue.

- When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter CS. The algorithm executes CS requests in the increasing order of timestamps.

Lamport's algorithm[3]



- Sites S_1 and S_2 are making requests for the CS
- Site S_2 enters the CS
- Site S_2 exits the CS and sends RELEASE messages
- Site S_1 enters the CS

The Ricart-Agrawala algorithm[1]

The Ricart-Agrawala algorithm is an optimization of Lamport's algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages. In this algorithm also, $\forall i : 1 \leq i \leq N : \mathbf{R_i} = \{S_1, S_2, \ldots, S_N\}$.

The Ricart-Agrawala algorithm[2]

Requesting the critical section.

1) When a site S_i wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.
2) When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS or if site S_j is requesting and S'_is request's timestamp is smaller than S_j's own request's timestamp. The request is deferred otherwise.

Executing the critical section

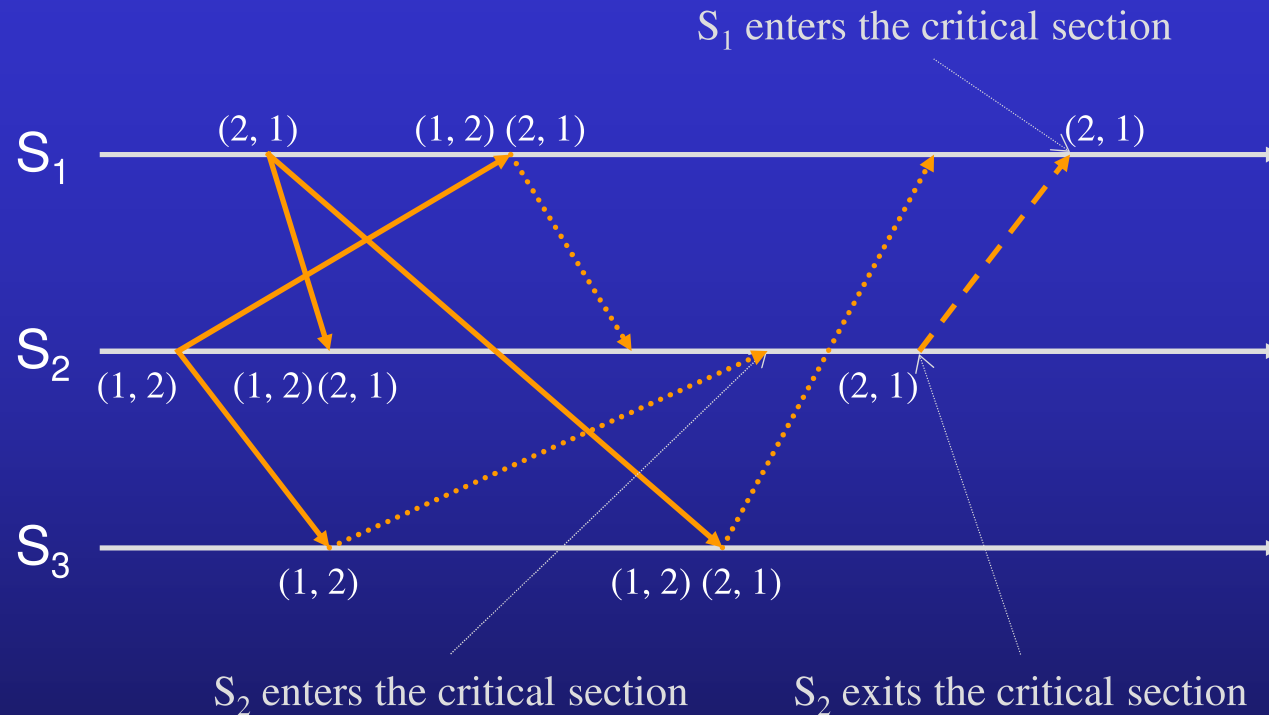
1) Site S_i enters the CS after it has received REPLY messages from all the sites in its request set.

Releasing the critical section

1) When site S_i exits the CS, it sends REPLY messages to all the deferred requests.

- A site's REPLY messages are blocked only by sites that are requesting the CS with higher priority (i.e., a smaller timestamp). Thus, when a site sends out REPLY messages to all the deferred requests, the site with the next highest priority request receives the last needed REPLY message and enters the CS. The execution of CS requests in this algorithm is always in the order of their timestamps.

The Ricart-Agrawala algorithm[3]



- Sites S_1 and S_2 are making requests for the CS
- Site S_2 enters the CS
- Site S_2 exits the CS and sends RELEASE messages
- Site S_1 enters the CS

Maekawa's algorithm[1]

Maekawa's algorithm is a departure from the general trend in the following two ways:

- First , a site does not request permission from every other site, but only from a subset of the sites. This is a radically different approach as compared to the Lamport and the Ricart-Agrawala algorithms, where all sites participate in the conflict resolution of all other sites. In Maekawa's algorithm the request set of sites are chosen such that $\forall i \, \forall j : 1 \leq i, j \leq N :: \mathbf{R_i} \cap \mathbf{R_j} \neq \Phi$. Consequently, every pair of sites has a site that mediates conflicts between that pair.
- Second, in Maekawa's algorithm a site can send out only one REPLY message at a time. A site can only send a REPLY message only after it has received a RELEASE message for the previous REPLY message. Therefore, a site S_i *locks* all the sites in R_i in exclusive mode before executing its CS.

Maekawa's algorithm[2]

The Construction of request sets. The request sets for sites in Maekawa's algorithm are constructed to satisfy the following conditions:

M1: $(\forall i\ \forall j : i \neq j,\ 1 \leq i, j \leq N :: \mathbf{R_i} \cap \mathbf{R_j} \neq \Phi)$.

M2: $(\forall i : 1 \leq i \leq N :: S_i \in \mathbf{R_i})$

M3: $(\forall i : 1 \leq i \leq N :: |\mathbf{R_i}| = K)$

M4: Any site Sj is contained in K number of $\mathbf{R_i}$s, $1 \leq i, j \leq N$.

Maekawa established the following relation between N and K:

$$N = K(K-1)+1.$$

This relation gives:

$$|\mathbf{R_i}| = \text{sqrt}(N).$$

Maekawa's algorithm[3]

Since there is at least one common site between the request sets of any two sites (condition **M1**), every pair of sites has a common site that mediates conflicts between the pair. A site can have only one outstanding REPLY message at any time; that is, it grants permission to an incoming request if it has not granted permission to some other site. Therefore, mutual exclusion is guaranteed. This algorithm requires the delivery of messages to be in the order they are sent between every pair of sites.

Conditions **M1** and **M2** are necessary for correctness, whereas conditions **M3** and **M4** provide other desirable features to the algorithm. Condition **M3** states that the size of the request sets of all the sites must be equal, implying that all sites should have to do an equal amount of work to invoke mutual exclusion. Condition **M4** enforces that exactly the same number of sites should request permission from any site, implying that all sites have responsibility in granting permission to other sites.

Maekawa's algorithm[4]

Requesting the critical section.

1) A site S_i requests access to the CS by sending REQUEST(*i*) messages to all the sites in its request set $\mathbf{R_i}$.
2) When a site S_j receives the REQUEST(*i*) message, it sends a REPLY(*j*) message to S_i provided it hasn't sent a REPLY message to a site from the time it received the last RELEASE message. Otherwise, it queues up the REQUEST for later consideration.

Executing the critical section.

1) Site S_i accesses the CS only after receiving REPLY messages from all the sites in $\mathbf{R_i}$.

Releasing the critical section.

1) After the execution of the CS is over, site S_i sends RELEASE(*i*) message to all the sites in $\mathbf{R_i}$.
2) When a site S_j receives a RELEASE(*i*) message from site S_i, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then site updates its state to reflect that the site has not sent out any REPLY message.

Suzuki-Kasami's broadcast algorithm[1]

In the Suzuki-Kasami's algorithm , if a site attempting to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all the other sites. A site that possesses the token sends it to the requesting site upon receiving its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has exited the CS. A site holding the token can enter its CS. A site holding the token can enter its CS repeatedly until it sends the token to some other site.

The main design issues in this algorithm are:

1) distinguishing outdated REQUEST messages from current REQUEST messages and
2) determining which site has an outstanding request for the CS.

Suzuki-Kasami's broadcast algorithm[2]

Outdated REQUEST messages are distinguished from current REQUEST messages in the following manner:

- A REQUEST message of site S_j has the form REQUEST(j, n), where n = 1, 2, ... is a sequence number that indicates that sites S_j is requesting its *nth* CS execution.
- A site S_i keeps an array of integers $RN_i[1..N]$, where $RN_i[j]$ is the largest sequence number received so far in a REQUEST message from site S_j.
- A REQUEST(j, n) message received by site S_i is outdated if $RN_i[j] > n$.
- When site S_i receives a REQUEST(j, n) message, it sets $RN_i[j]:=\max(RN_i[j], n)$.

Suzuki-Kasami's broadcast algorithm[2]

- Sites with outstanding requests for the CS are determined using the token contents.
- The token is composed of ***token queue*** *Q* and ***token array*** *LN*, where *Q* is a queue of requesting nodes and *LN* is an array of size N, such that *LN*[*j*] is the sequence number of the request that site S_j executed most recently.

Suzuki-Kasami's broadcast algorithm[2]

- After executing its CS, a site S_i updates $LN[i]:=RN_i[i]$ to indicate that its request corresponding to sequence number $RN_i[i]$ has been executed.
- The token array $LN[1..N]$ permits a site to determine if some other site has an outstanding request for the CS.
- Note that at site S_i if $RN_i[j]=LN[j]+1$, then site S_j is currently requesting the token.
- After having executed the CS, a site checks this condition for all the j's to determine all the sites that are requesting the token and places their ids in queue Q if not already present in this queue Q.
- Then the site sends the token to the site at the head of the queue Q.

Suzuki-Kasami's broadcast algorithm[3]

Requesting the critical section.

1) If the requesting site S_i does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a REQUEST(i, sn) message to all other sites (sn is the updated value of $RN_i[i]$).
2) When a site Sj receives this message, it sets $RN_j[i]$ to max($RN_j[i]$, sn). If S_j has the idle token, then it sends the token to S_i if $RN_j[i]=LN[i]+1$.

Suzuki-Kasami's broadcast algorithm[3]

Executing the critical section.

1) Site S_i executes the CS when it has received the token.

Suzuki-Kasami's broadcast algorithm[3]

Releasing the critical section. Having finished the execution of the CS, site S_i takes the following actions:

1) It sets $LN[i]$ element of the token array equal to $RN_i[i]$.
2) For every site S_j whose ID is not in the token queue, it appends its ID to the token queue if $RN_i[j]=LN[j]+1$.
3) If token queue is nonempty after the above update, then it deletes the top site ID from the queue and sends the token to the site indicated by the ID.

- The Suzuki-Kasami algorithm is not symmetric because a site retains the token even if it does not have a request for the CS, which is contrary to the spirit of Ricart and Agrawala's definition of a symmetric algorithm: "*no site possesses the right to access its CS when it has not been requested.*"

Raymond's tree-based algorithm[1]

In Raymond's tree-based algorithm, sites are logically arranged as a directed tree such that the edges of the tree are assigned directions toward the site (root of the tree) that has the token. Every site has a local variable ***holder*** that points to an immediate neighbor node on a directed path to the root node. Thus, *holder* variables at the sites define logical tree structure among the sites. If we follow *holder* variables at sites, every site has a directed path leading to the site holding the token. At root site, *holder* points to itself.

Every site keeps a FIFO queue, called ***request_q***, which stores the requests of those neighboring sites that have sent a request to this site, but have not yet been sent the token.

Raymond's tree-based algorithm[2]

Requesting the critical section.

1) When a site wants to enter the CS, it sends a REQUEST message to the node along the directed path to the root, provided it does not hold the token and its *request_q* is empty. It then adds its request to its *request_q*. (Note that a nonempty *request_q* at a site indicates that the site has sent a REQUEST message to the root node for the top entry in its *request_q*).
2) When a site recives a REQUEST message, it places the REQUEST in its *request_q* and sends a REQUEST message along the directed path to the root provided it has not sent out a REQUEST message on its outgoing edge (for a previously received REQUEST on its *request_q*).
3) When the root site receives a REQUEST message, it sends the token to the site from which it received the REQUEST message and sets its *holder* variable to point at that site.
4) When a site receives the token, it deletes the top entry from its *request_q,* sends the token to the site indicated in this entry, and sets its *holder* variable to point at that site. If the *request_q* is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by *holder* variable.

Raymond's tree-based algorithm[3]

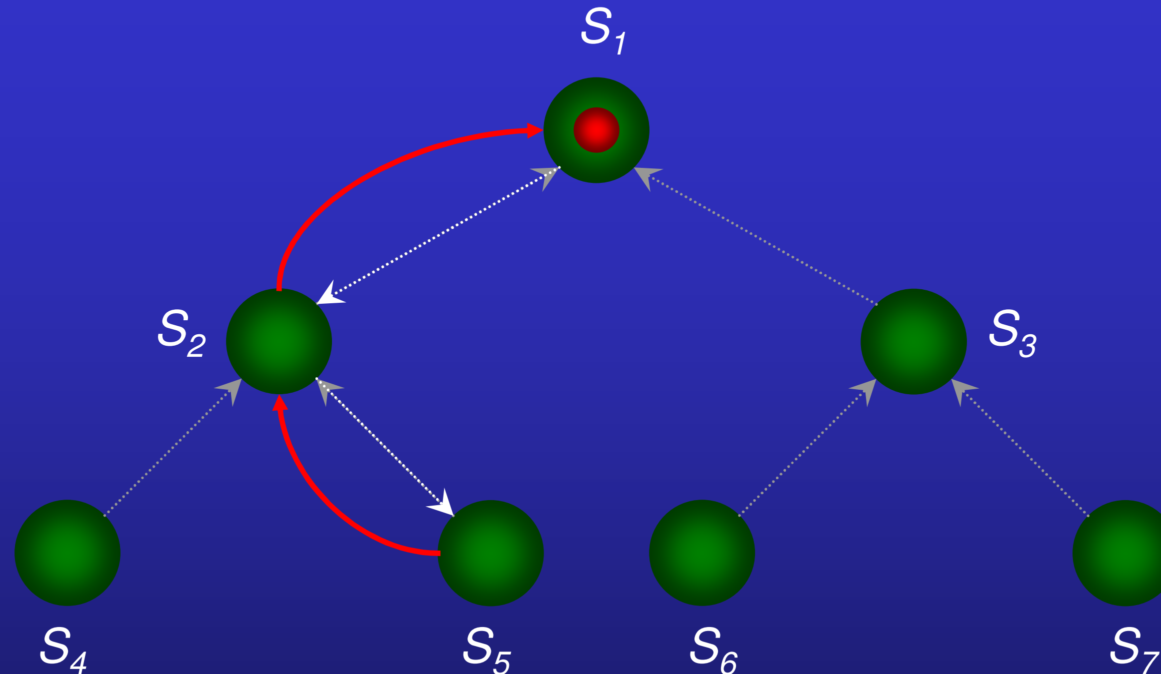
Executing the critical section.

1) A site enters the CS when it receives the token and its own entry is at the top of its *request_q*. In this case, the site deletes the top entry from its *request_q* and enters the CS.

Releasing the critical section. After a site has finished execution of the CS, it takes the following actions:

1) If its *request_q* is nonempty, then it deletes the top entry from its *request_q*, sends the token to that site, and sets its *holder* variable to point at that site.
2) If the *request_q* is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by the *holder* variable.

Raymond's tree-based algorithm[4]



- Sites arranged in a tree configuration.
- Site S_5 is requesting the token.
- The token is in transit to S_5.
- State after S_5 has received the token.

Election algorithms

- Bully algorithm
- Ring algorithm

Bully algorithm

When a process notices that the coordinator is no longer responding to requests, it initiates an election.

1) Process *P* sends an ELECTION message to all processes with higher numbers.
2) If no one responds, *P* wins the election and becomes coordinator.
3) If one of the higher-ups answers, it takes over. *P's* job is done.

At any moment, a process can get an ELECTION message from one of its lower numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

Ring algorithm

The algorithm uses a ring, but without a token. The processes are physically or logically ordered.

1) When any process notices that coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor, and so on.
2) Eventually, the message gets back to the process that started it all. The message type is changed to COORDINATOR and circulated once again, this time to inform everyone who the coordinator is (the list member with the highest number).

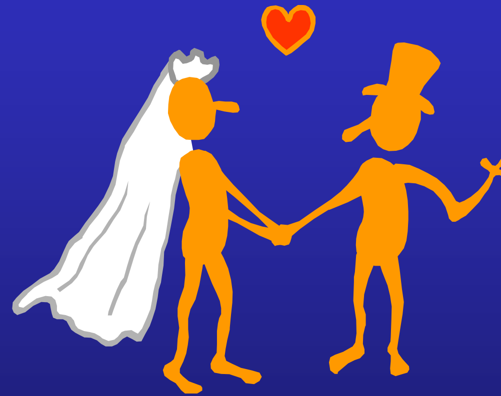
…

and

they lived

happily

ever after



The End