TIME AND STATE IN DISTRIBUTED SYSTEMS

Time in Distributed Systems

☞ Because each machine in a distributed system has its own clock there is no notion of *global physical time*.

- The *n* crystals on the *n* computers will run at slightly different rates, causing the clocks gradually to get out of synchronization and give different values.

Problems:

- Time triggered systems: these are systems in which certain activities are scheduled to occur at predefined moments in time. If such activities are to be coordinated over a distributed system we need a coherent notion of time.

  Example: time-triggered real-time systems

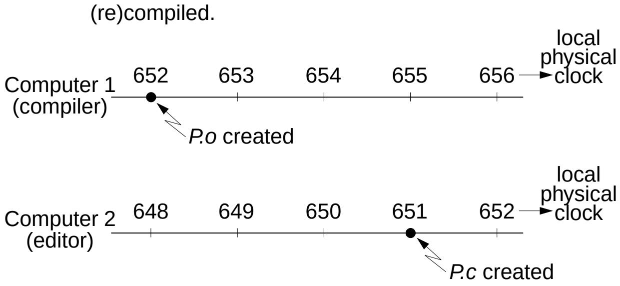
- Maintaining the consistency of distributed data is often based on the *time* when a certain modification has been performed.

  Example: a *make* program.

Time in Distributed Systems (cont'd)

The *make*-program example

- When the programmer has finished changing some source files he starts *make*; *make* examines the times at which all object and source files were last modified and decides which source files have to be (re)compiled.



Although *P.c* is modified after *P.o* has been generated, because of the clock drift the time assigned to *P.c* is smaller.

⇩

P.c will not be recompiled for the new version!

Time in Distributed Systems (cont'd)

Solutions:

☞ **Synchronization of physical clocks**

- Computer clocks are synchronized with one another to an achievable, known, degree of accuracy ⇒ within the bounds of this accuracy we can coordinate activities on different computers using each computer's local clock.
- Physical clock synchronization is needed for distributed real-time systems.

☞ **Logical clocks**

- In many applications we are not interested in the physical time at which events occur; what is important is the *relative order* of events!

  The *make*-program is such an example (slide 3).

- In such situations we don't need synchronized physical clocks.
  Relative ordering is based on a virtual notion of time - *logical time*.
- Logical time is implemented using *logical clocks*.

Lamport's Logical Clocks

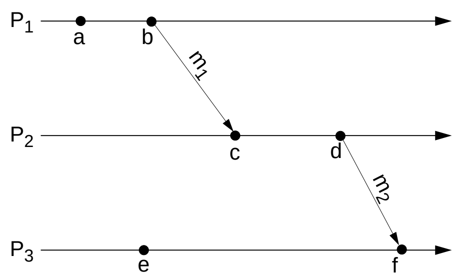
☞ The order of events occurring at different processes is critical for many distributed applications.
Example: *P.o_created* and *P.c_created* in slide 3.

☞ Ordering can be based on two simple situations:
1. If two events occurred in the same process then they occurred in the order observed following the respective process;
2. Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving it.

☞ Ordering by Lamport is based on the *happened-before relation* (denoted by $\rightarrow$):
- $a \rightarrow b$, if a and b are events in the same process and a occurred before b;
- $a \rightarrow b$, if a is the event of sending a message m in a process, and b is the event of the same message m being received by another process;
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (the relation is transitive).

Lamport's Logical Clocks (cont'd)

- If $a \rightarrow b$, we say that event a causally affects event b. The two events are causally related.
- There are events which are not related by the *happened-before* relation.
If *both* a $\rightarrow$ e *and* e $\rightarrow$ a *are false*, then a and e are concurrent events; we write $a \parallel e$.



P1, *P2*, *P3*: processes;
a, *b*, *c*, *d*, *e*, *f*: events;

$a \rightarrow b$, $c \rightarrow d$, $e \rightarrow f$, $b \rightarrow c$, $d \rightarrow f$
$a \rightarrow c$, $a \rightarrow d$, $a \rightarrow f$, $b \rightarrow d$, $b \rightarrow f$, ...
$a \parallel e$, $c \parallel e$, ...

Lamport's Logical Clocks (cont'd)

☞ Using physical clocks, the happened before relation can not be captured. It is possible that $b \rightarrow c$ and at the same time $T_b > T_c$ (T_b is the physical time of b).

☞ *Logical clocks* can be used in order to capture the *happened-before* relation.

- A logical clock is a monotonically increasing software counter.
- There is a logical clock C_{Pi} at each process P_i in the system.
- The value of the logical clock is used to assign *timestamps* to events.
$C_{Pi}(a)$ is the timestamp of event a in process P_i.
- There is no relationship between a logical clock and any physical clock.

To capture the happened-before relation, logical clocks have to be implemented so that
if $a \rightarrow b$, then $C(a) < C(b)$

Lamport's Logical Clocks (cont'd)

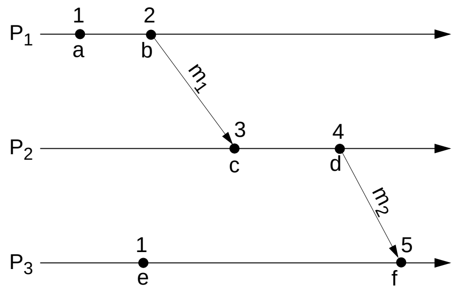
☞ Implementation of logical clocks is performed using the following rules for updating the clocks and transmitting their values in messages:

[R1]: C_{Pi} is incremented before each event is issued at process P_i: $C_{Pi} := C_{Pi} + 1$.

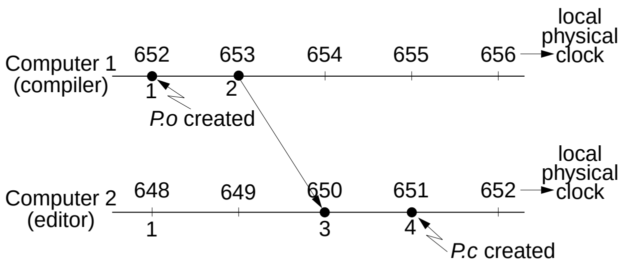
[R2]: a) When a is the event of sending a message m from process P_i, then the timestamp $t_m = C_{Pi}(a)$ is included in m ($C_{Pi}(a)$ is the logical clock value obtained after applying rule R1).
b) On receiving message m by process P_j, its logical clock C_{Pj} is updated as follows:
$C_{Pj} := \max(C_{Pj}, t_m)$.
c) The new value of C_{Pj} is used to timestamp the event of receiving message m by P_j (applying rule R1).

- If a and b are events in the same process and a occurred before b, then $a \rightarrow b$, and (by R1) $C(a) < C(b)$.
- If a is the event of sending a message m in a process, and b is the event of the same message m being received by another process, then $a \rightarrow b$, and (by R2) $C(a) < C(b)$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$, and (by induction) $C(a) < C(c)$.

Lamport's Logical Clocks (cont'd)



- For the *make*-program example we suppose that a process running a compilation notifies, through a message, the process holding the source file about the event *P.o created* $\Rightarrow$ a logical clock can be used to correctly timestamp the files.



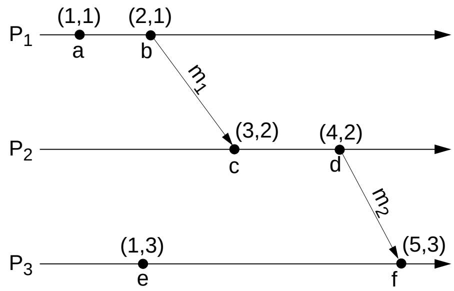
Problems with Lamport's Logical Clocks

☞ Lamport's logical clocks impose only a partial order on the set of events; pairs of distinct events generated by *different* processes can have identical timestamp.

- For certain applications a total ordering is needed; they consider that no two events can occur at the same time.
- In order to enforce total ordering a *global logical timestamp* is introduced:
  - the global logical timestamp of an event *a* occurring at process P_i, with logical timestamp $C_{Pi}(a)$, is a pair ($C_{Pi}(a)$, i), where i is an identifier of process P_i;
  - we define

    ($C_{Pi}(a)$, i) < ($C_{Pj}(b)$, j) if and only if

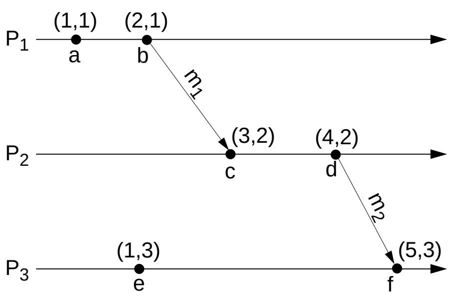
    $C_{Pi}(a) < C_{Pj}(b)$, or $C_{Pi}(a) = C_{Pj}(b)$ and $i < j$.



Problems with Lamport's Logical Clocks (cont'd)

☞ Lamport's logical clocks are not powerful enough to perform a causal ordering of events.

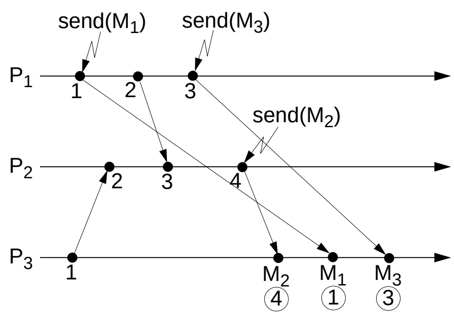
- if $a \rightarrow b$, then $C(a) < C(b)$.
  However, the reverse is not always true (if the events occurred in different processes):
  if $C(a) < C(b)$, then $a \rightarrow b$ is **not necessarily** true.
  (it is only guaranteed that $b \rightarrow a$ is not true).



$C(e) < C(b)$, however there is no causal relation from event *e* to event *b*.

- By just looking at the timestamps of the events, we cannot say whether two events are causally related or not.

Problems with Lamport's Logical Clocks (cont'd)



- We would like messages to be processed according to their causal order.
  We would like to use the associated *timestamp* for this purpose.
- Process P_3 receives messages M_1, M_2, and M_3.
  $send(M_1) \rightarrow send(M_2)$, $send(M_1) \rightarrow send(M_3)$,
  $send(M_3) \parallel send(M_2)$
- M_1 has to be processed before M_2 and M_3.
  However P_3 has not to wait for M_3 in order to process it before M_2 (although M_3's *logical clock timestamp* is smaller than M_2's).

Vector Clocks

☞ Vector clocks give the ability to decide whether two events are causally related or not by simply looking at their timestamp.

- Each process P_i has a clock C^v_{Pi}, which is an integer vector of length n (n is the number of processes).
- The value of C^v_{Pi} is used to assign timestamps to events in process P_i.
  $C^v_{Pi}(a)$ is the timestamp of event a in process P_i.
- $C^v_{Pi}[i]$, the ith entry of C^v_{Pi}, corresponds to P_i's own logical time.
- $C^v_{Pi}[j]$, $j \neq i$, is P_i's "best guess" of the logical time at P_j.
  $C^v_{Pi}[j]$ indicates the (logical) time of occurrence of the last event at P_j which is in a *happened-before* relation to the current event at P_i.

Vector Clocks (cont'd)

☞ Implementation of vector clocks is performed using the following rules for updating the clocks and transmitting their values in messages:

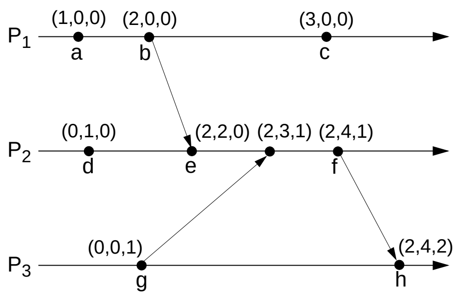
[R1]: C^v_{Pi} is incremented before each event is issued at process P_i: $C^v_{Pi}[i] := C^v_{Pi}[i] + 1$.

[R2]: a) When a is the event of sending a message m from process P_i, then the timestamp $t_m = C^v_{Pi}(a)$ is included in m ($C^v_{Pi}(a)$ is the vector clock value obtained after applying rule R1).

b) On receiving message m by process P_j, its vector clock C^v_{Pj} is updated as follows:
$\forall k \in \{1,2,..,n\}$, $C^v_{Pj}[k] := \max(C^v_{Pj}[k], t_m[k])$.

c) The new value of C^v_{Pj} is used to timestamp the event of receiving message m by P_j (applying rule R1).

Vector Clocks (cont'd)



For any two vector timestamps u and v, we have:
- $u = v$ if and only if $\forall i$, $u[i] = v[i]$
- $u \leq v$ if and only if $\forall i$, $u[i] \leq v[i]$
- $u < v$ if and only if $(u \leq v \wedge u \neq v)$
- $u \parallel v$ if and only if $\neg(u < v) \wedge \neg(v < u)$

☞ Two events a and b are causally related if and only if $C^v(a) < C^v(b)$ or $C^v(b) < C^v(a)$. Otherwise the events are concurrent.

☞ With vector clocks we get the property which we missed for Lamport's logical clocks:
- $a \rightarrow b$ if and only if $C^v(a) < C^v(b)$.
  Thus, by just looking at the timestamps of the events, we can say whether two events are causally related or not.

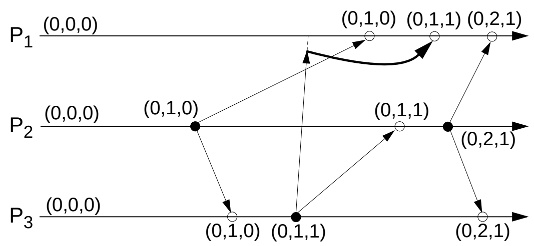
Causal Ordering of Messages Using Vector Clocks

The problem has been formulated on slide 12:

☞ We would like messages to be processed according to their causal order.

- If $Send(M_1) \rightarrow Send(M_2)$, then every recipient of both messages M_1 and M_2 must receive M_1 before M_2.

Causal Ordering of Messages Using Vector Clocks (cont'd)



☞ A message delivery protocol which preforms causal ordering based on vector clocks.

- Basic Idea:
  - A message is delivered to a process only if the message immediately preceding it (considering the causal ordering) has been already delivered to the process. Otherwise, the message is buffered.
- We assume that processes communicate using broadcast messages.
  (There exist similar protocols for non-broadcast communication too.)

Causal Ordering of Messages Using Vector Clocks (cont'd)

- The events which are of interest here are the sending of messages ⇒ vector clocks will be incremented only for message sending.

☞ Implementation of the protocol is based on the following rules:

[R1]: a) Before broadcasting a message *m*, a process P_i increments the vector clock: $C^v_{Pi}[i] := C^v_{Pi}[i] + 1$.
b) The timestamp $t_m = C^v_{Pi}$ is included in *m*.

[R2]: The receiving side, at process P_j, delays the delivery of message *m* coming from P_i until both the following conditions are satisfied:
1. $C^v_{Pj}[i] = t_m[i] - 1$
2. $\forall k \in \{1,2,..,n\} - \{i\}$, $C^v_{Pj}[k] \geq t_m[k]$

Delayed messages are queued at each process in a queue that is sorted by their vector timestamp; concurrent messages are ordered by the time of their arrival.

[R3]: When a message is delivered at process P_j, *its vector clock* C^v_{Pj} is updated according to rule R2b for vector clock implementation (see slide 14).

☞ $t_m[i] - 1$ indicates how many messages originating from P_i precede *m*.
Step R2.1 ensures that process P_j has received all the messages originating from P_i that precede *m*.
Step R2.2 ensures that P_j has received all those messages received by P_i before sending *m*.

Global States

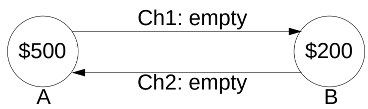
☞ The problem is how to collect and record a *consistent global state* in a distributed system.

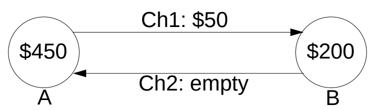
Why a problem?

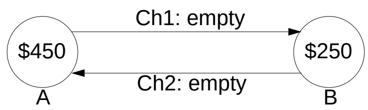
- Because there is no global clock (no coherent notion of time) and no shared memory!

Global States (cont'd)

Consider a bank system with two accounts *A* and *B* at two different sites; we transfer $50 between *A* and *B*.







A	Ch1	B	C : consistent NC: not consistent
500	empty	200	C
500	50	200	NC
450	50	200	C
450	empty	200	NC
500	50	250	NC
450	50	250	NC
450	empty	250	C
500	empty	250	NC

Global States (cont'd)

☞ In general, a global state consists of a set of local states and a set of states of the communication channels.

☞ The state of the communication channel *in a consistent global state* should be the sequence of messages sent along the channel before the sender's state was recorded, excluding the sequence of messages received along the channel before the receiver's state was recorded.

☞ It is difficult to record channel states to ensure the above rule ⇒ *global states are very often recorded without using channel states*.
This is the case in the definition below.

Formal Definition

- LS_i is the local state of process P_i.
  Beside other information, the local state also includes a record of all messages sent and received by the process.
- We consider the global state GS of a system, as the collection of the local states of its processes:
  $GS = \{LS_1, LS_2, ..., LS_n\}$.
- A certain global state can be consistent or not!

- $send(m_{ij}^k)$ denotes the event of sending message m_{ij}^k from P_i to P_j;
  $rec(m_{ij}^k)$ denotes the event of receiving message m_{ij}^k by P_j.

- $send(m_{ij}^k) \in LS_i$ if and only if the sending event occurred before the local state was recorded;
  $rec(m_{ij}^k) \in LS_j$ if and only if the receiving event occurred before the local state was recorded.

- $transit(LS_i,LS_j) = \{m_{ij}^k \mid send(m_{ij}^k) \in LS_i \wedge rec(m_{ij}^k) \notin LS_j\}$
  $inconsistent(LS_i,LS_j) = \{m_{ij}^k \mid send(m_{ij}^k) \notin LS_i \wedge rec(m_{ij}^k) \in LS_j\}$

Formal Definition (cont'd)

☞ A global state $GS = \{LS_1, LS_2, ..., LS_n\}$ is *consistent* if and only if:

$$\forall i, \forall j: 1 \le i, j \le n :: inconsistent(LS_i,LS_j) = \varnothing$$

- In a consistent global state for every received message a corresponding send event is recorded in the global state.
- In an inconsistent global state, there is at least one message whose receive event is recorded but its send event is not recorded.

☞ A global state $GS = \{LS_1, LS_2, ..., LS_n\}$ is *transitless* if and only if:

$$\forall i, \forall j: 1 \le i, j \le n :: transit(LS_i,LS_j) = \varnothing$$

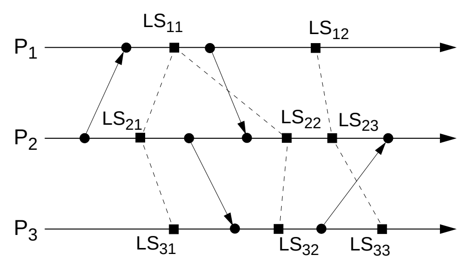
- All messages recorded to be sent are also recorded to be received.

☞ A global state is *strongly consistent* if it is consistent and transitless.

- A strongly consistent state corresponds to a consistent state in which all messages recorded as sent are also recorded as received.

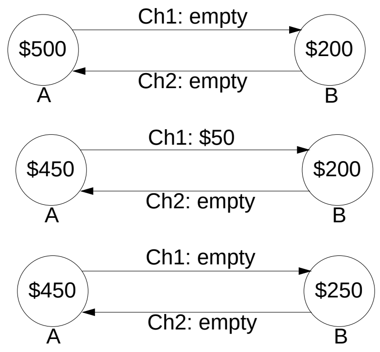
Note: the global state, as defined here, is seen as a collection of the local states, without explicitely capturing the state of the channel.

Formal Definition (cont'd)



$\{LS_{11}, LS_{22}, LS_{32}\}$ is inconsistent;
$\{LS_{12}, LS_{23}, LS_{33}\}$ is consistent;
$\{LS_{11}, LS_{21}, LS_{31}\}$ is strongly consistent.

Formal Definition (cont'd)



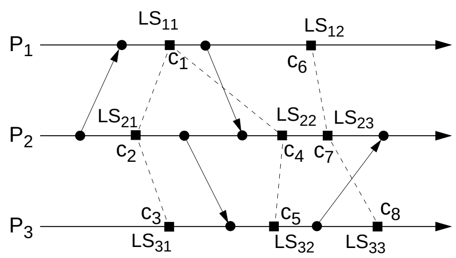
A	B	C : consistent NC: not consistent
500	200	{A,B}: strongly C
450 (mess$_1$ sent)	200	{A,B}: C
500	250 (mess$_1$ received)	{A,B}: NC
450 (mess$_1$ sent)	250 (mess$_1$ received)	{A,B}: strongly C

- After registering of the receive event(s) a consistent state becomes strongly consistent. It is considered to be a normal (transient) situation.

Cuts of a Distributed Computation

☞ A *cut* is a graphical representation of a global state. A *consistent cut* is a graphical representation of a consistent global state.

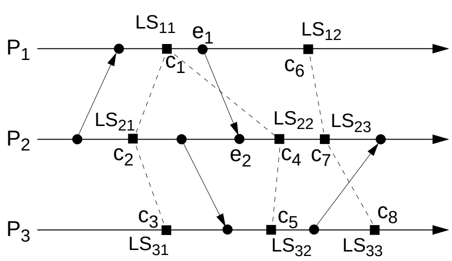
- A cut of a distributed computation is a set $Ct = \{c_1, c_2, ..., c_n\}$, where c_i is the cut event at process *Pi*.
- A cut event is the event of recording a local state of the respective process.



Cuts of a Distributed Computation (cont'd)

☞ Let e_k denote an event at process P_k.
A cut $Ct = \{c_1, c_2, ..., c_n\}$ is a consistent cut if and only if
$\forall P_i, \forall P_j, \nexists e_i, \nexists e_j$ such that $(e_i \rightarrow e_j) \wedge (e_j \rightarrow c_j) \wedge \neg(e_i \rightarrow c_i)$

- A cut is consistent if every message that was received before a cut event was sent before the cut event at the sender process.



$\{c_1, c_4, c_5\}$ is not consistent: $(e_1 \rightarrow e_2) \wedge (e_2 \rightarrow c_4) \wedge \neg(e_1 \rightarrow c_1)$

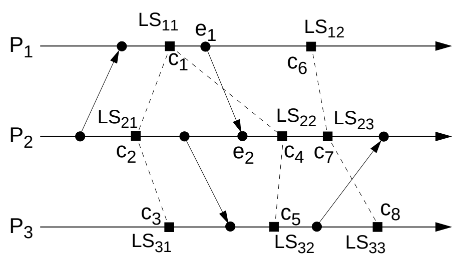
Cuts of a Distributed Computation (cont'd)

Theorem

A cut $Ct = \{c_1, c_2, ..., c_n\}$ is a consistent cut if and only if no two cut events are causally related, that is:

$$\forall c_i, \forall c_j :: \neg(c_i \rightarrow c_j) \wedge \neg(c_j \rightarrow c_i)$$

- A set of concurrent cut events form a consistent cut.



$\{c_1,c_2,c_3\}$: strongly consistent (no communication line is crossed)

$\{c_6,c_7,c_8\}$: consistent (communication line is crossed but no causal relation).

$\{c_1,c_4,c_5\}$: not consistent; $c_1 \rightarrow c_4$

Global State Recording
(Chandy-Lamport Algorithm)

- The algorithm records a collection of local states which give a consistent global state of the system. *In addition it records the state of the channels which is consistent with the collected global state*.
- Such a recorded "view" of the system is called a *snapshot*.
- We assume that processes are connected through one directional channels and message delivery is FIFO.
- We assume that the graph of processes and channels is strongly connected (there exists a path between any two processes).
- The algorithm is based on the use of a special message, *snapshot token*, in order to control the state collection process.

Global State Recording (cont'd)

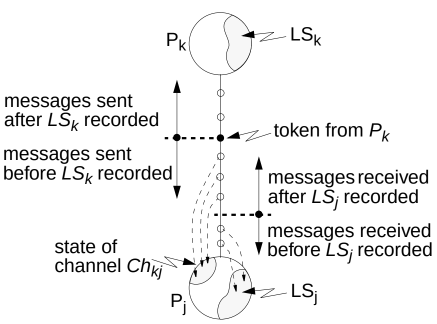
Some discussion on how to collect a global state:

- A process P_i records its local state LS_i and later sends a message *m* to P_j; LS_j at P_j has to be recorded *before* P_j has received *m*.
- The state SCh_{ij} of the channel Ch_{ij} consists of all messages that process P_i sent before recording LS_i and which have not been received by P_j when recording LS_j.
- A snapshot is started at the request of a particular process P_i, for example, when it suspects a deadlock because of long delay in accessing a resource; P_i then records its state LS_i and, before sending any other message, it sends a token to every P_j that P_i communicates with.
- When P_j receives a token from P_i, and this is the first time it received a token, it must record its state before it receives the next message from P_i. After recording its state P_j sends a token to every process it communicates with, before sending them any other message.

Global State Recording (cont'd)

What about the channel states?

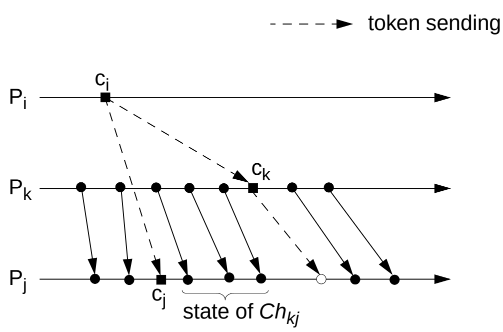
- P_i sends a token to P_j and this is the first time P_j received a token ⇒ P_j immediately records its state. All the messages sent by P_i before sending the token have been received at P_j ⇒ $SCh_{ij} := \varnothing$.
- P_j receives a token from P_k, but P_j already recorded its state. *M* is the set of messages that *Pj* received from P_k after P_j recorded its state and before P_j received the token from P_k ⇒ $SCh_{kj} := M$.



- The algorithm terminates when all processes have received tokens on all their input channels.
- The process that initiated the snapshot should be informed; it can collect the global snapshot.

Global State Recording (cont'd)

Maybe, you prefer this view:



- Don't forget when you look to the picture: we assumed that message passing on a channel connecting two processes is FIFO.

Global State Recording (cont'd)

The algorithm

☞ Rule for sender P_i:

/* performed by the initiating process and by any other process at the reception of the first token */

[SR1]: P_i records its state.

[SR2]: P_i sends a token on each of its outgoing channels.

☞ Rule for receiver P_j:

/* executed whenever P_j receives a token from another process P_i on channel Ch_{ij} */

[RR1]: **if** P_j has not yet recorded its state **then**

Record the state of the channel: $SCh_{ij} := \varnothing$.

Follow the "*Rule for sender*".

else

Record the state of the channel: $SCh_{ij} := M$, where M is the set of messages that Pj received from P_i after P_j recorded its state and before P_j received the token on Ch_{ij}.

end if.

Summary

- In a distributed system there is no exact notion of global physical time. Physical clocks can be synchronized to a certain accuracy.
- In many applications not physical time is important but only the relative ordering of certain events. Such an ordering can be achieved using logical clocks.
- Lamport's logical clocks are implemented using a monotonic integer counter at each site. They can be used in order to capture the happened-before relation.
- The main problem with Lamport's clocks is that they are not powerful enough to perform a causal ordering of events.
- Vector clocks give the ability to decide whether two events are causally related or not, by simply looking at their timestamps.

Summary (cont'd)

- As there doesn't exist a global notion of physical time, it is very difficult to reason about a global state in a distributed system.
- We can consider a global state as a collection of local states and, possibly, a set of states of the communication channels.
- A global state can be consistent or not.
- A cut is a graphical representation of a global state. Using cuts it is easy to elegantly reason about consistency of global states.
- It is possible to record local states and states of the channels, so that together they provide a consistent view of the system. Such a view is called a snapshot.