Process-Process Synchronization

- The Coordinated Attack Problem
- Partial Ordering of Events in a Distributed System
- Vector Clocks
- Logical Clocks
- Total Ordering of Events in a Distributed System
- Physical Clock Synchronization
- Snapshot Algorithm

Decentralized Control of Processes

- A distributed system without centralized control
- Distributed resource allocation
- Synchronization of processes
- Give a global order to the different events in the system
- To achieve a similar view of the system global state

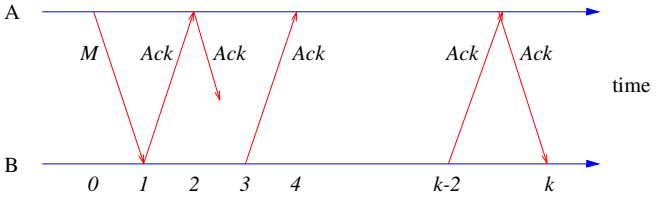
Synchronization of processes

The Coordinated Attack Problem

Presumption: Two processes communicate using messages on an unreliable medium where messages can be corrupted or disappear.

Statement: It is impossible for the two processes to agree on a specific point of time when both should change their states simultaneously.

Proof: here we will show that it is not possible even in a simplified case when the messages have a constant network transmission time (if the message arrives).



If we can agree at time *k* then the last message can not be part of the decision since process A does not know if it has arrived or not. Then the decision could as well be done at time *k-1*. But then process B does not know if its last message has arrived or not so it can not be part of the decision. Then the decision could as well be done at time *k-2*., and so forth. We will find the no message can be part of the decision and then no decision can be made.

Synchronization of processes

The Coordinated Attack Problem

Consequences

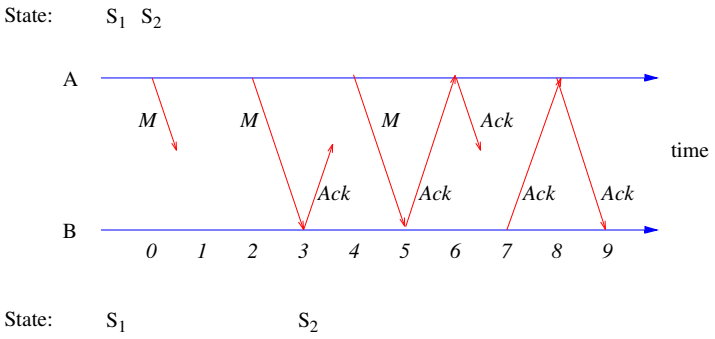
- It is impossible to keep a distributed database in a consistent state at any moment of time.
- Automatic Teller Machine (ATM)
  - Can not deliver money at the same moment as it is drawn from the corresponding account.
- Bank account systems
  - It is impossible to move money from an account in one bank to another momentarily.
- In a distributed computer system it is not possible to transfer a resource from one computer to another momentarily.
- It is hard to control a distributed real time system.
- Y. Yemini, D. Cohen, 1979.

Synchronization of processes

The Coordinated Attack Problem

What is possible

- It is possible for two processes to decide that they change their states **some time** in the future.



- The system **"converges"** to a consistent state.

ATM

- ATM account withdrawal:
  - First it sends a message to the bank system about card number and the withdrawal amount.
  - The bank system checks that the withdrawal is OK and then draws the amount from the account.
  - Then a message is sent back to the ATM that it can output the money.
  - When the message reaches the ATM it outputs the money bills.
- Should there arise a communication breakdown during the transaction the system might remain in an inconsistent state:
  - The amount is drawn from the account but no money has been output.
- This will be corrected in due time when the communication is working again.

Typical communication protocol

- Sender sends a message
  - Now only the sender knows about the message.
- Receiver accepts message, sends an acknowledgement
  - Now both sender and receiver knows about the message.
  - Only the receiver knows that is has gone through.
- The sender get acknowledgement
  - Now both sender and receiver knows about the message.
  - Now both sender and receiver knows that is has gone through.
- Sender sends next message

Synchronization of processes

Consequences 2

- It is impossible for the processes in a distributed system to know the **global** state of the system without stopping it.

Ordering the Events in a Distributed System

- Leslie Lamport, 1978.
- Presumptions:
  - A number of processes are communicating using message passing in an unreliable network. Each process is a sequence of events.
  - These events (in the same process) are totally ordered in time.
  - A message passing will have an unknown transmission time > 0.
  - The sending and the receiving of messages will be seen as the events in a process.

Partial Ordering of Events

"$\rightarrow$" (happens) before

Definition: "$\rightarrow$" on the set of events in a distributed system is the least relation that fulfills:

(*i*) If a and b are two events in the same process and a happens before b, $a \rightarrow b$ holds.

(*ii*) If a is the sending of a message in one process and b is the receiving of the same message in another process, then $a \rightarrow b$ holds.

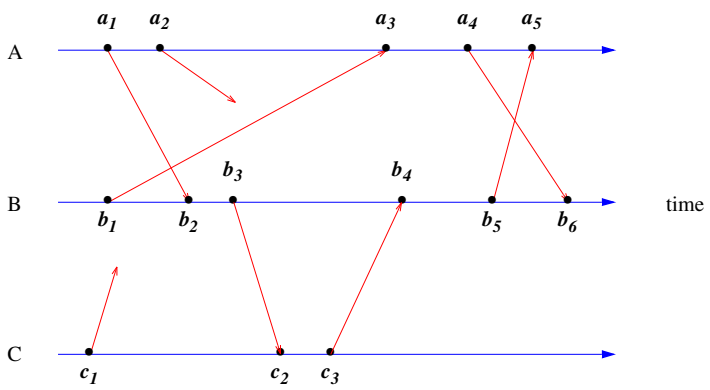
(*iii*) If both $a \rightarrow b$ and $b \rightarrow c$ holds, then $a \rightarrow c$ holds.

(*iv*) If an event a "not happens before" an event b it is denoted as $a \not\rightarrow b$.

(*v*) If $a \not\rightarrow b$ and $b \not\rightarrow a$ holds, then a and b in a way are "concurrent", this is denoted as $a \,//\, b$:

- "$\rightarrow$" is an irreflexive partial order on the set of events in a system.

Events in a system from a global viewpoint



$a_1 \rightarrow a_2 \quad a_1 \rightarrow b_2 \quad a_1 \rightarrow c_3$ since $a_1 \rightarrow b_2, b_2 \rightarrow b_3, b_3 \rightarrow c_2$ and $c_2 \rightarrow c_3$

$a_2 \not\rightarrow b_3$ and $b_3 \not\rightarrow a_2$ gives $a_2 \,//\, b_3$
$a_2 \not\rightarrow b_2$ and $b_2 \not\rightarrow a_2$ gives $a_2 \,//\, b_2$

but $b_2 \rightarrow b_3$

Vector Clocks

- A vector (array) is used to represent the partial ordering "$\rightarrow$".
- Each process holds a clock vector $C_i\,[1..n]$
  n is the number of processes in the system.

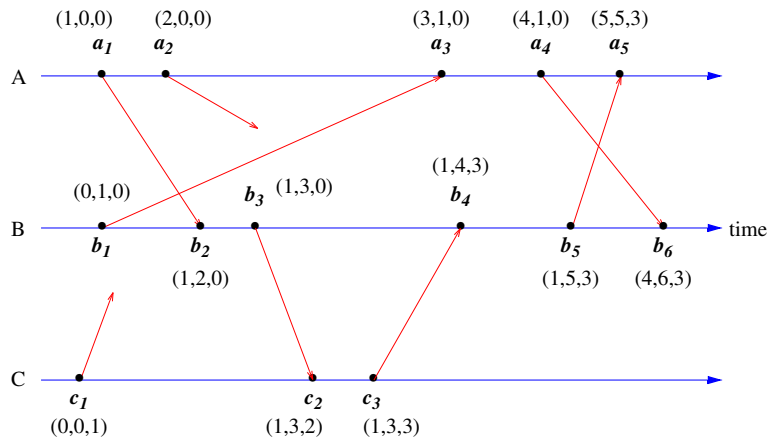
  It is defined as:
  - $C_i\,[i]$ describes the time in process P_i.
  - $C_i\,[j]$ describes process P_i's knowledge about the time in process P_j. Updated each time P_i gets a message from P_j.
  - Each event in P_i is time stamped with the entire vector $C_i\,[1..n]$. This represents P_i's global view of the event.
- This leads to the following rules for each process P_i:

  (i) Each process P_i increments its local register $C_i\,[i]$ for each event.

  (ii) Each message is stamped with the actual **full** clock vector $C_i\,[1..n]$.

  (iii) When receiving a message a process should adjust its local clock value, $C_i\,[1..n]$, comparing it to the message timestamp $T\,[1..n]$ according to:
  $1 \le k \le n$: $C_i\,[k] := \max(C_i\,[k], T\,[k])$,
  then $C_i\,[i]$ is incriminated.

Vector Clock Example



Implementing Partial Order using Vector Clocks

- The partial ordering, "→", can be calculated by comparing the vector clock values.
- Definitions:

  (i) If $C_i \leq C_j \Leftrightarrow \forall x: C_i[x] \leq C_j[x]$
  vector C_j is said to dominate vector C_i.

  (ii) If $C_i < C_j \Leftrightarrow C_i \leq C_j \wedge \exists x: C_i[x] < C_j[x]$
  vector C_j is said to strictly dominate vector C_i.

  (iii) If $C_i \,//\, C_j \Leftrightarrow \neg(C_i < C_j) \wedge \neg(C_j < C_i)$
  vector C_i and vector C_j is said to be "parallel".

- Then it holds for two events a and b with vector timestamps C_a and C_b:

  $a \rightarrow b \Leftrightarrow C_a < C_b$

  $a \,//\, b \Leftrightarrow C_a \,//\, C_b$

- Note that this is a partial ordering, there will be events a and b with vectors like (iii) above. Sometimes we say we have a conflict when this happens.

Use of Vector Clocks

- Distributed debugging.
- In distributed memory.
- Definition of global breakpoints.
- Keep a consistent distributed file system with multiple copies at network partitions.
- Measure of distributed algorithms amount of parallelism.

Logical Clocks

- Leslie Lamport, 1978.

A **Logical Clock** is a function that assigns an ordering number $C(a)$ to an event a in the system.

Clock condition: $\forall a,b$: a and b are events in the system:
$a \rightarrow b \Rightarrow C(a) < C(b)$.

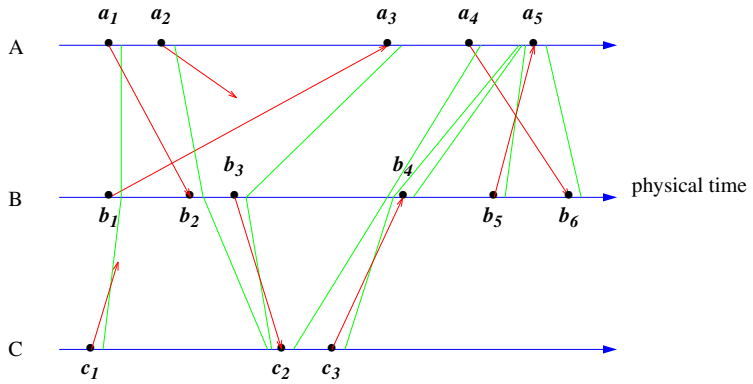
(N.B.! We don't require the reverse)

- Each process P_i has an en instance C_i of the logical clock
- If a is an event in process $P_i \Rightarrow C(a) = C_i(a)$
- If a and b are events in process P_i and $a \rightarrow b \Rightarrow C_i(a) < C_i(b)$
  - the clock must step forward (tick) at least by 1 between two events in the same process.
- If a is the sending of a message by process P_i and b is the receiving of the same message by process $P_j \Rightarrow C_i(a) < C_j(b)$
  - a message must be received at a clock value that is higher than the sending clock value

Implementing Logical Clocks

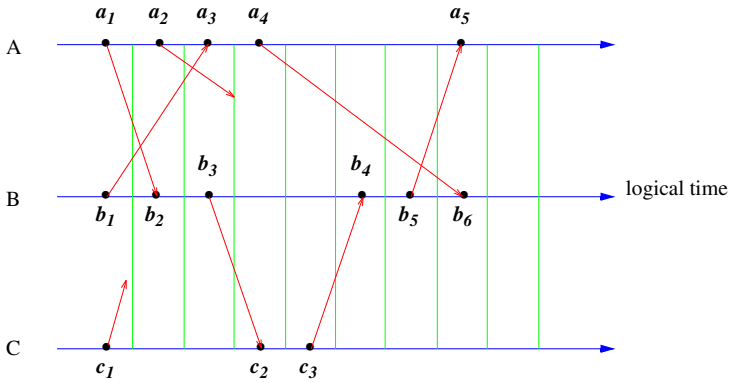
- Each process P_i increments its local register C_i after each event in the process.
- Each message contains a timestamp T_m.
  - If a is the event that process P_i sends a message m, then m will contain the timestamp $T_m = C_i(a)$.
- When a process receives a message it first compares its timestamp to the local logical clock register.
  - Before a process P_j registers the reception of a message m, P_j will set its local logical clock register $C_j := max(C_j, T_m+1)$ which will be the logical clock value for the message receiving event.

Logical Clock example as seen globally



- This diagrams shows the events when they really happen in the "real" physical time.
  This can not be known by the processes.
  Red lines indicate messages with time stamps that are sent among the processes.
- The green lines shows the "ticks" of the Logical Clock.
  This means that a_2 and b_2 have the same Logical Clock value.
  Also a_4 and c_2 have the same Logical Clock value.

Logical Clock example as seen by the processe



- Since the processes do not know the physical time they have to use the Logical Clock values, i.e. they will have to follow the green lines.
  - For the processes the green lines will be thought of as vertical.
  - The events might be in different order than according to real time, but if for two events, a and b, that $a \rightarrow b$ holds then the logical clock value for a must be lower than for b.

Total ordering of events

Introduce an ordering relation "$\prec$" on the processes:
$P_i \prec P_j$, means P_i is before P_j

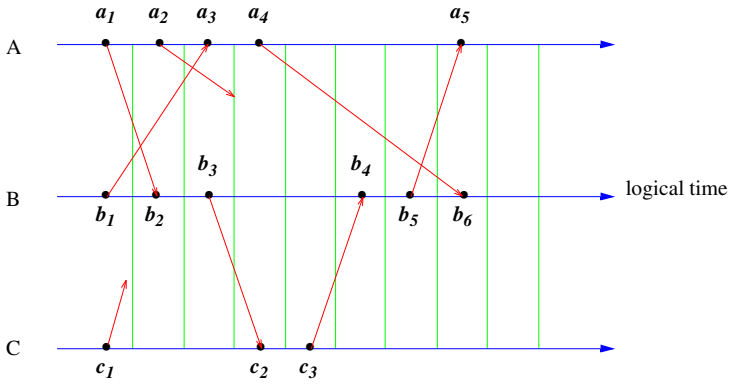
Definition: "$\rightarrow_T$"
Let a be an event in process P_i and b be an event in process P_j:

(*i*) if $C_i(a) < C_j(b)$ then $a \rightarrow_T b$.

(*ii*) if $C_i(a) = C_j(b)$ and $P_i \prec P_j$ then $a \rightarrow_T b$.

- "$\rightarrow_T$" is a total ordering of the events in a system where processes communicate using message passing in a computer network:
  - $\neg(a \rightarrow_T b) \Rightarrow (b \rightarrow_T a)$
  - $a \rightarrow b \Rightarrow a \rightarrow_T b$
- "$\prec$" will introduce priorities on the processes

Total ordering example



Assume A ≀ B ≀ C.

Then the following holds:
$a_1 \rightarrow_T b_1 \rightarrow_T c_1 \rightarrow_T a_2 \rightarrow_T b_2 \rightarrow_T a_3 \rightarrow_T b_3 \rightarrow_T a_4 \rightarrow_T c_2 \rightarrow_T c_3 \rightarrow_T b_4 \rightarrow_T \rightarrow_T b_5 \rightarrow_T a_5 \rightarrow_T b_6$

Distributed Resource Allocation Algorithm using Logical Clock

- Algorithm conditions:
  - The algorithm uses
    - Distributed Request Queue (empty at start)
    - Logical Clock
  - each process administers:
    - a local copy of the Request Queue
    - a local copy of the Logical Clock
    - a table containing the latest received timestamp from each of the other processes in the system (TS-table)
  - When placing a request at process P_i, it is associated with the current local logical clock value C_i in P_i.
  - all messages between two processes is delivered in the same order as they were sent and no message will disappear (FIFO).
    This will be the case when using a communication protocol such as TCP/IP.

Algorithm

- The different events:
  - P_i *REQUEST*:
    - $<T_i;P_i;REQUEST>$ is sent to all other processes, T_i is the timestamp taken from C_i
    - $<T_i;P_i;REQUEST>$ is also put in P_i's local copy of the Request Queue sorted according to "$\rightarrow_T$"
    - P_i increments its local Logical Clock value
  - P_j receives *REQUEST* $<T_i;P_i;REQUEST>$:
    - P_j adjusts its local copy of the Logical Clock according to the Logical Clock definition
    - $<T_i;P_i;REQUEST>$ is put in P_j's local copy of the Request Queue sorted according to "$\rightarrow_T$"
    - P_j updates its TS-table
    - P_j increments its local Logical Clock value
    - P_j sends an acknowledgement $<T_j;P_j;ACK>$ to P_i
    - P_j increments its local Logical Clock value
  - P_i receives an acknowledgement $<T_j;P_j;ACK>$ from P_j:

- P_i adjusts its local copy of the Logical Clock according to the Logical Clock definition.
- P_i updates its TS-table with T_j from P_j
- P_i increments its local Logical Clock value
  - P_i is allowed access to the resource when:
    - $<T_i;P_i;REQUEST>$ is number one in the (local) Request Queue
    - $T_i \rightarrow_T T_j$ for all T_j in the (local) TS-table
  - P_i want to *RELEASE*:
    - $<T_i';P_i;RELEASE>$ is sent to all other processes, T_i' is the timestamp taken from actual C_i
    - $<T_i;P_i;REQUEST>$ is erased from the (local) Request Queue
    - P_i increments its local Logical Clock value
  - P_j receives $<T_i';P_i;RELEASE>$:
    - P_j adjusts its local copy of the Logical Clock according to the Logical Clock definition.
    - $<T_i;P_i;REQUEST>$ is erased from the (local) Request Queue
    - P_j updates its TS-table with T_i' from P_i
    - P_j increments its local Logical Clock value

Totally ordered Multicast

- The algorithm can be used for sending totally ordered Multicast Messages.
- Instead of sending Requirement messages, Multicast messages with logical clock values can be sent and ordered according to the total order in a message queue.
- The messages are delivered to the receiving process following the total order definition.

Physical Clocks

Why?

- Let S stand for **all** events in the system,
  not only messages in the network but also other interactions
- "$\rightarrow_a$" is defined as happens before in S
- How avoid abnormal behavior?

  (i) require that all interaction should involve exchange of timestamps

  (ii) use synchronized physical clocks

Strong clock condition: $\forall \boldsymbol{a},\boldsymbol{b} \in S: \boldsymbol{a} \rightarrow_a \boldsymbol{b} \Rightarrow C(\boldsymbol{a}) < C(\boldsymbol{b})$

Clock Conditions

- Each clock must have high precision, i.e. have correct speed:

  $\exists \kappa << 1: \forall i: \left|\frac{d}{dt}C_i(t) - 1\right| < \kappa$

  Typical value for a modern clock might be $\kappa < 10^{-6}$.

- The value difference between two clocks should not be too large,
  i.e. the clocks should be synchronized:

  $\forall i,j: |C_i(t) - C_j(t)| < \varepsilon$ where ε is the maximal difference between any two clocks.

  The clocks must show the "same" time.

- Let μ denote the shortest time for a process to influence another process

For a correct behavior of the system it is required that
$\varepsilon / (1 - \kappa) \leq \mu$, since then $C_i(t+\mu) - C_j(t) > 0$
This requirement will assure that every message will experience that time is moving forward

- consequently $\varepsilon \leq \mu(1-\kappa)$ must hold

- One solution would be to send clock values between the processes to set clock values as close as possible

Setting clock values using messages between nearby processes

- We use the following notation:

  μ: the minimum time to send a message between two processes

  ν_m: the real time to send a message m

  ξ_m: the den unpredictable delay of a message m

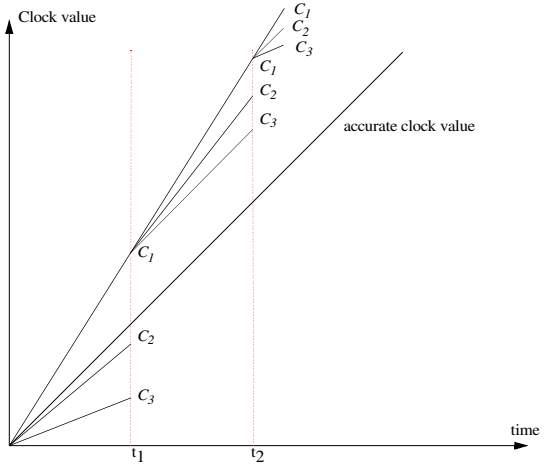
  then

  $\xi_m = \nu_m - \mu$. for message m

Clock adjusting algorithm

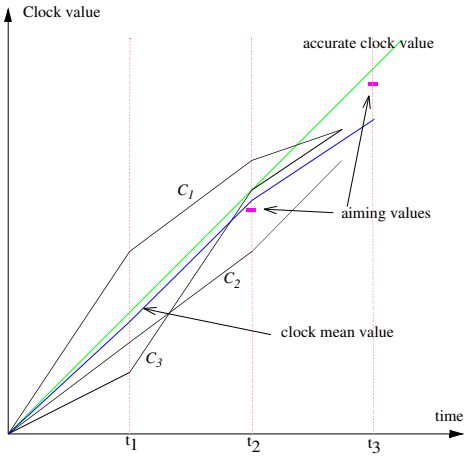
- Leslie Lamport, 1978.
  - If process P_i doesn't get a message at time t it lets its local clock C_i step forward using its internal time keeper.
  - If process P_i sends a message m at time t, this message will contain a timestamp $T_m = C_i(t)$
  - If process P_i receives a message m at time t containing a timestamp T_m P_i should set its local clock according to:
    $C_i := max(C_i, T_m + \mu_m)$
- A network with diameter D (the maximal number of the minimum number of node jumps between any two nodes in the network).
- Each node sends a message to each neighbor at least every τ time units.
  - Then the clock difference will be maximized by
    $\varepsilon \approx D(2\kappa\tau + \xi)$
    - The problem here is that theoretically this is unlimited since ξ is unlimited.
- The fastest clock will push the other clocks forward.

Physical Clocks Example 1



- At times t_1 and t_2 the clocks, C_1, C_2 and C_3 compare their values.
  Since no clock is allowed to be changed backwards they all change to the highest value.
  - This means that the fastest clock will decide the time

Physical Clocks Example 2

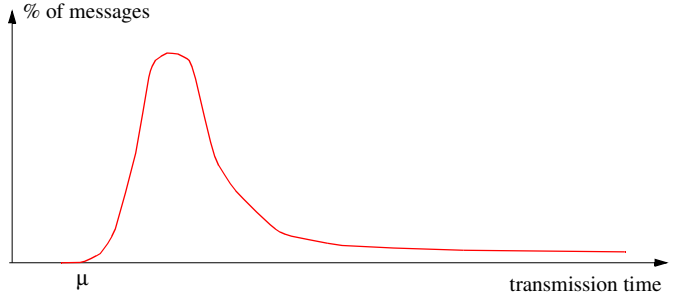


- Instead of taking the highest value as the new value, the mean value of the clocks is chosen..
  Since no clock is allowed to be changed backwards they change their speeds in order to reach the next calculated value, the aiming value.
  The blue line shows what is supposed to be the accurate time of the system.

Probabilistic Clock Synchronizing

Characteristics to take into consideration when constructing a better algorithm

- Most messages have short transmission time.
- but at least a minimal time, μ

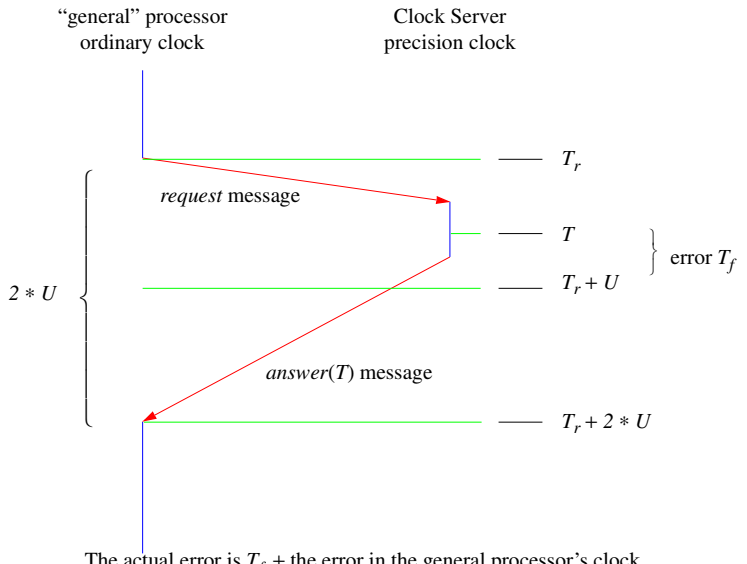


Probabilistic Clock Synchronizing

- Flaviu Cristian, 1989.
- To set the clients clock a time request message is sent to the clock server.
  - Total waiting time from sending request until receiving reply is $2 * U$
  - The received time value is T.
    It is assumed that this time was read by the server after half the waiting time U.
    Then the estimated time when receiving the reply will be $T + U$.
- Calculaing the error:
  - The server can only have read this time between the sending of the request and the receiving of the reply. Thus the error can be estimated to be $\pm U$.
  - Since there is a minimum transmission time, μ, for the messages this error estimation can be refined to $\pm (U - \mu)$
  - But we have to read our own clock to get U. Since the clock might have a speed error κ we have to add $\kappa * U$ to the error estimathion

Thus the time can be estimated as $T + U \pm (U - \mu + \kappa * U)$

Probabilistic Clock Synchronizing



The actual error is T_f + the error in the general processor's clock

Snapshot Algorithm

Chandy and Lamport, 1985.

- Calculates a "useful" approximate global state of a distributed system, although it might never have existed.
- Algorithm conditions:
  - the nodes are part of a directed graph, real or virtual
  - the graph is strongly connected, i.e. there is a path from any node to any other node
  - the algorithm messages are only sent according to the directed graph
  - FIFO secure transmissions on the communication links (e.g. TCP/IP)
  - the process can continue working and change their states while the snapshot takes place
- Calculates an estimated global state of the system consistent with the real state.
  - this might never have existed
  - but this state is a possible state that doesn't conflict with the real states and might be used for system control.

Consistent State

- Assume that S_1 is the real global state when the algorithm starts and S_2 is the real state when the algorithm terminates.
- A calculated state S_c is consistent with the real state if
  - $S_c = S_1$, or
  - $S_c = S_2$, or
  - $S_1 \rightarrow S_c \rightarrow S_2$ is a possible state sequence.

Snapshot-algorithm

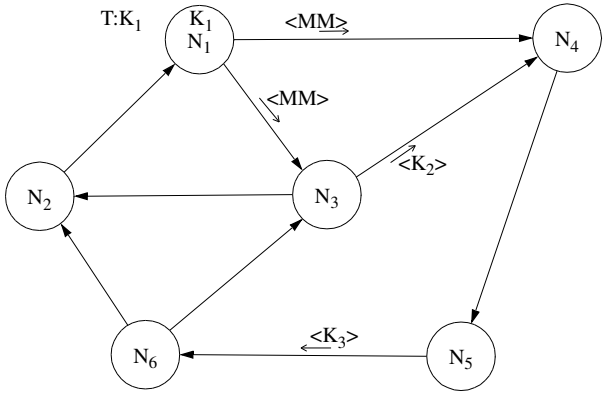
- uses *Marker Messages* (*MM*)
- *Local Calculated State* (*LS*)
  - A node initiates the algorithm
    - records its local state, *LS*
    - puts itself in recording state
    - sends an *MM*-message on each out-going link
  - When a node gets its first *MM*-message on an in-going link
    - records its local state, *LS*
    - puts itself in recording state
    - marks the in-going link as ready
    - sends an *MM*-message on each out-going link
  - When a node in recording state gets an *MM*-message on an in-going link
    - marks the in-going link as ready

  - When a node in recording state gets another message on an in-going link
    - if the in-going link is not marked as ready the recorded state, *LS,* will be recalculated according to the message
    - if the in-going link is marked as ready the message will not affect the calculated state
  - When a node in recording state has got an *MM*-message on all in-going links
    - its part of the global calculated state, *LS,* is ready.
    - its recorded state is
      - sent to one node for assembling the global calculated state
      - or sent to all other nodes so all can calculate the global state
    - it leaves recording state
  - A calculating node that has got the local calculated states, *LS,* from all other nodes can put them together to the Calculated Global State
- The Calculated Global State might not have been existing for the system.

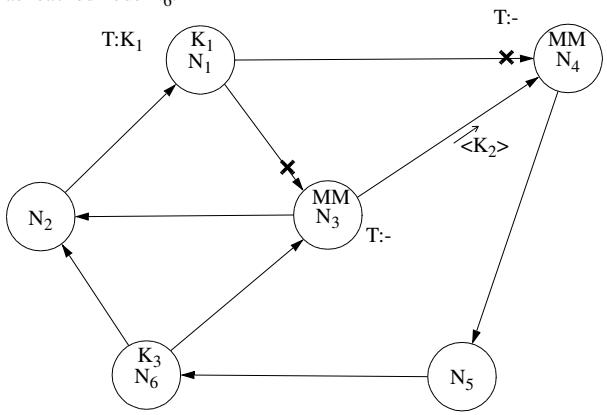
Example: Counting Capabilities

- A Capability is a permission (grant) ticket to be used in a system for access to different resources.
  - the CORBA ior-string
  - the right to use a certain program (license)
  - the right to read/change a file
  - a Control Token
- In a distributed system a capability can be sent in a message between processes.
- In this example we assume that the number of capabilities in the system should be constant (e.g. we have paid a license for a program that allows n users in the same time.
- A capability might get lost or one process might not delete its own copy when sending it to another process.
  - We need an algorithm to count the capabilities in the system at one moment
  - We don't want to stop the system to do this
  - Then we can use the Snapshot Algorithm

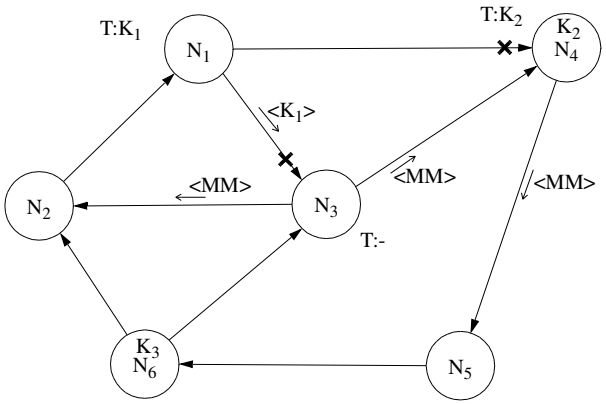
- Nodes N_1, N_2, N_3, N_4, N_5 and N_6
- Capabilities K_1, K_2 and K_3.
- Node N_1 starts the algorithm by recording K_1 and then sending out MM on outgoing links.
- Capabilities K_2 and K_2 are being transferred among nodes.



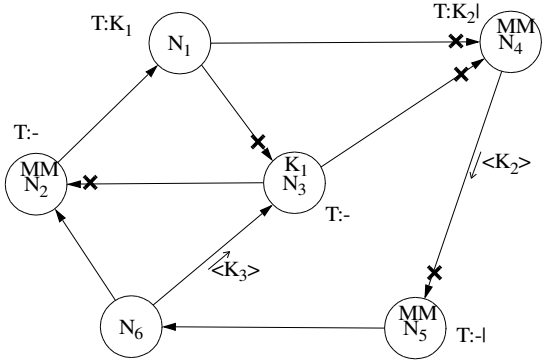
- When node N_3 and node N_4 received the MM-messages they had no capabilities. They both record empty.
- Capability K_3 has reached node N_6.



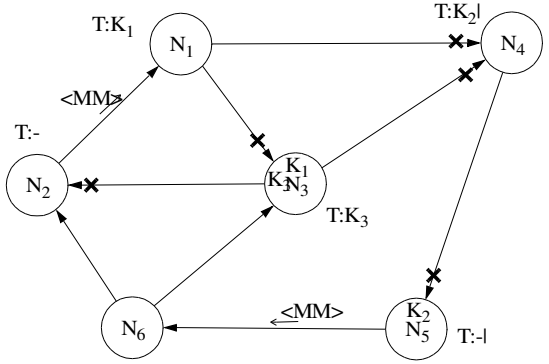
- The nodes N_3 and N_4 sends MM-messages on their outgoing links.
- Capability K_2 then reached N_4 and is recorded since N_4 has not got an MM-message on the corresponding link yet.
- Also node N_1 has sent capability K_1 in a message in its way to N_3



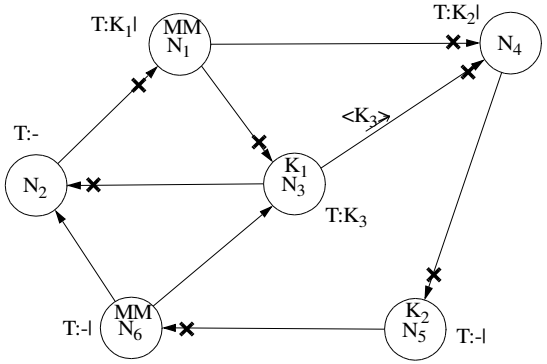
- The MM-messages have now reached nodes N_2, N_4 and N_5.
- Since node N_4 now has got MM-messages on all in links it is ready with its part of the algorithm.
  For the global state N_4 contributes with K_2.
  Nodes N_2 and N_5 record empty
- Nodes N_2 and N_5 have not got an MM before so they send an MM-message on each outgoing link.
  Node N_5 has got MM-message on all in links and is ready.
- Capability K_1 has reached node N_3. Since N_3 already has got an MM-message on the corresponding link it does not record K_1 as a part of its contribution to the algorithm.



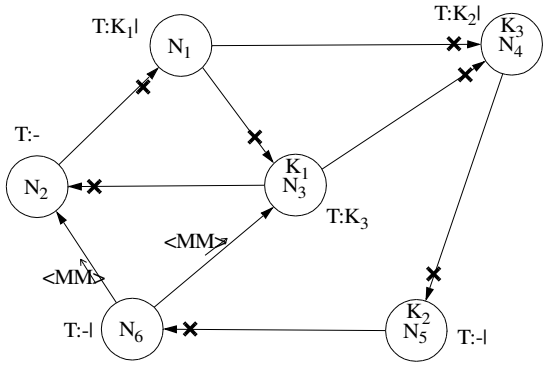
- Capabilities K_3 and K_2 have been sent.
- Since the nodes N_2 and N_5 have not got an MM-message before they send MM-messages on all their outgoing links.
- K_2 and K_3 have reached N_5 and N_3
  Since N_5 has already got an MM-message on the corresponding link K_2 is not recorded by N_5.
  K_3 is on the other side recorded by node N_3 since N_3 has not got an MM-message on the corresponding link.



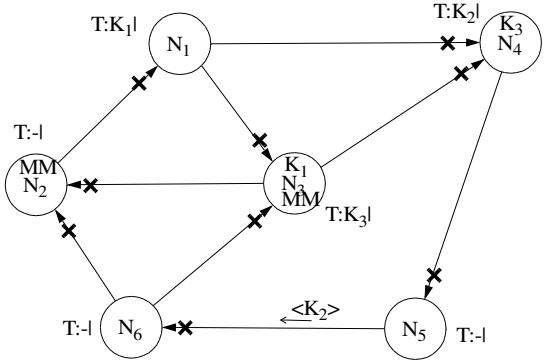
- The MM-message from N_2 has reached N_1 and the MM-message from N_5 has reached N_6.
- Node N_1 then is ready with its part of the algorithm.
- Node N_6 gets its first MM-message and records its state which is empty.



- N_6 then sends MM-messages on its outgoing links.
- K_3 has arrived to N_4.



- K_2 has been sent from N_5 to N_6.
- The MM-messages have reached N_2 and N_3.
  Now all nodes have got MM-messages on all in links and the local calculations are ready.



- By letting one or all nodes combine the local recorded states a global state will be calculated.
- This state might never existed in reality but can still be used for the distributed control.
  - In our example it gives the correct number of capabilities in the system although it does not tell where they actually are.
- The calculated state: