Chapter 11. Time and Global States

11.1 Introduction

11.2 Clocks, events and process states

11.3 Synchronizing physical clocks

11.4 Logical time and logical clocks

11.5 Global states

11.1 Introduction

- We need to measure time accurately:
  - to know the time an event occurred at a computer
  - to do this we need to synchronize its clock with an authoritative external clock
- Algorithms for clock synchronization useful for
  - concurrency control based on timestamp ordering
  - authenticity of requests e.g. in Kerberos
- There is no global clock in a distributed system
  - this chapter discusses clock accuracy and synchronization
- Logical time is an alternative
  - It gives ordering of events - also useful for consistency of replicated data

Ch 2: Computer clocks and timing events

- Each computer in a DS has its own internal clock
  - used by local processes to obtain the value of the current time
  - processes on different computers can timestamp their events
  - but clocks on different computers may give different times
  - computer clocks drift from perfect time and their drift rates differ from one another.
  - ***clock drift rate***: the relative amount that a computer clock differs from a perfect clock
- Even if clocks on all computers in a DS are set to the same time, their clocks will eventually vary quite significantly unless corrections are applied

11.2 Clocks, events and process states

How to order the events that occur at a single processor

- A distributed system is defined as a collection *P* of *N* processes p_i, $i = 1,2,\ldots N$
- Each process p_i has a state s_i consisting of its variables (which it transforms as it executes)
- Processes communicate only by messages (via a network)
- Actions of processes*:*
  - *Send, Receive,* change own state
- *Event:* the occurrence of a single action that a process carries out as it executes e.g. *Send, Receive,* change state
- Events at a single process p_i, can be placed in a total ordering denoted by the relation $\rightarrow_i$ between the events. i.e.

  $e \rightarrow_i e'$ if and only if the event e occurs before e' at p_i
- A history of process p_i : is a series of events ordered by $\rightarrow_i$.

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

Clocks

How to timestamp the events that occur at a single processor

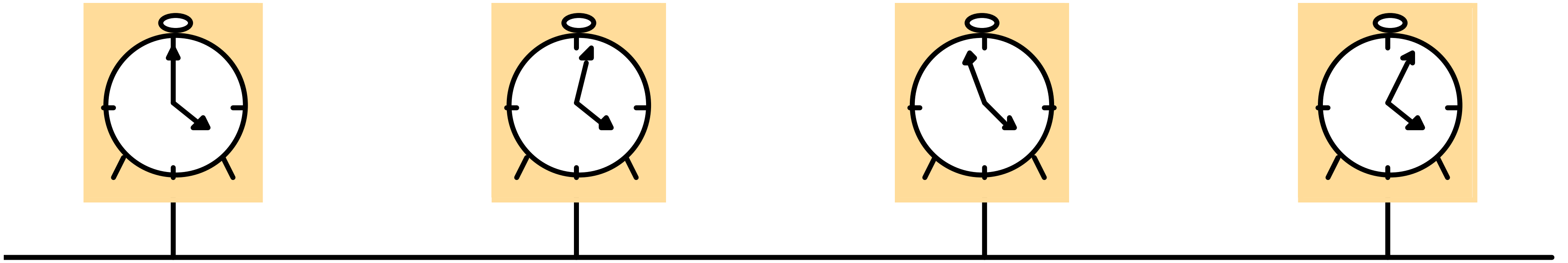
- How to assign to them a date and time of day
- To timestamp events, use the computer's clock
- At real time, *t*, the OS reads the time on the computer's hardware clock $H_i(t)$
- It calculates the time on its software clock

  $$C_i(t) = \alpha H_i(t) + \beta$$

  - . if C_i behaves well enough, we can use its value to timestamp any event at p_i

- Successive events will correspond to different timestamps only if the clock resolution < time interval between successive events
- Clock resolution: the period between updates of the clock value

Skew between computer clocks in a distributed system



Network

- Computer clocks are not generally in perfect agreement
- *Skew*: the difference between the times on two clocks (at any instant)
- Computer clocks are subject to *clock drift* (they count time at different rates)
- Clock *drift rate*: the difference per unit of time from some ideal reference clock
- Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
- High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Coordinated Universal Time (UTC)

- International Atomic Time is based on very accurate physical clocks (drift rate 10^{-13})
- UTC is an international standard for time keeping
- It is based on atomic time, but occasionally adjusted to astronomical time
- It is broadcast from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1-10 millisecond
- Signals from GPS are accurate to about 1 microsecond

Why can't we put GPS receivers on all our computers?

11.3 Synchronizing physical clocks

- External synchronization
  - A computer's clock C_i is synchronized with an external authoritative time source S, so that:
  - $|S(t) - C_i(t)| < D$ for $i = 1, 2, \dots N$ over an interval, I of real time
  - The clocks C_i are accurate to within the bound D.
- Internal synchronization
  - The clocks of a pair of computers are synchronized with one another so that:
  - $|C_i(t) - C_j(t)| < D$ for $i, j = 1, 2, \dots N$ over an interval, I of real time
  - The clocks C_i and C_j agree within the bound D.
- Internally synchronized clocks are not necessarily externally synchronized, as they may drift collectively
- if the set of processes P is synchronized externally within a bound D, it is also internally synchronized within bound $2D$

Clock correctness

- A hardware clock, *H* is said to be correct if its drift rate is within a bound $\rho > 0$. (e.g. 10^{-6} secs/ sec)
- This means that the error in measuring the interval between real times *t* and *t'* is bounded:
  - $(1 - \rho)\ (t' - t) \leq H(t') - H(t) \leq (1 + \rho)\ (t' - t)$ (where $t'>t$)
  - Which forbids jumps in time readings of hardware clocks
- Weaker condition of monotonicity may suffice:
  - $t' > t \Rightarrow C(t') > C(t)$ that is, a clock *C* only ever advances
  - can achieve monotonicity with a hardware clock that runs fast by adjusting the values of α and β in $C_i(t) = \alpha H_i(t) + \beta$
- *a faulty* clock is one that does not obey its correctness condition
- *crash failure* - a clock stops ticking
- *arbitrary* failure - any other failure e.g. jumps in time

Synchronization in a synchronous system

- a synchronous distributed system is one in which the following bounds are defined (ch. 2):
  - the time to execute each step of a process has known lower and upper bounds
  - each message transmitted over a channel is received within a known bounded time
  - each process has a local clock whose drift rate from real time has a known bound

- Internal synchronization in a synchronous system
  - One process p_1 sends its local time t to process p_2 in a message m,
  - p_2 could set its clock to $t + T_{trans}$ where T_{trans} is the time to transmit m
  - T_{trans} is unknown but $min \le T_{trans} \le max$
    - min can be measured or conservatively estimated
    - max known in synchronous system
  - uncertainty $u = max\text{-}min$. Set clock to $t + (max - min)/2$ then skew $\le u/2$

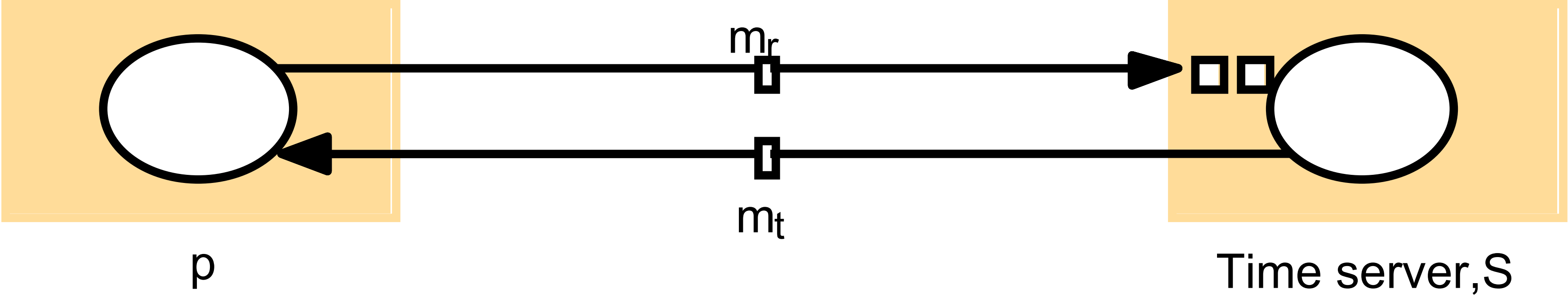
Cristian’s method (1989) for an asynchronous system

- External synchronization
- A time server *S* receives signals from a UTC source
  - Process *p* requests time in m_r and receives *t* in m_t from *S*
  - *p* sets its clock to $t + T_{round}/2$
  - Accuracy ± $(T_{round}/2 - min)$:

T_{round} is the round trip time recorded by *p*
min is an estimated minimum round trip time

Wrong! min should be one way trip time!

    - because the earliest time *S* puts *t* in message m_t is *min* after *p* sent m_r
    - the latest time was *min* before m_t arrived at *p*
    - the time by *S*’s clock when m_t arrives is in the range $[t+min, t + T_{round} - min]$
    - the width of the range is $T_{round} - 2min$, so the accuracy is ± $(T_{round}/2 - min)$



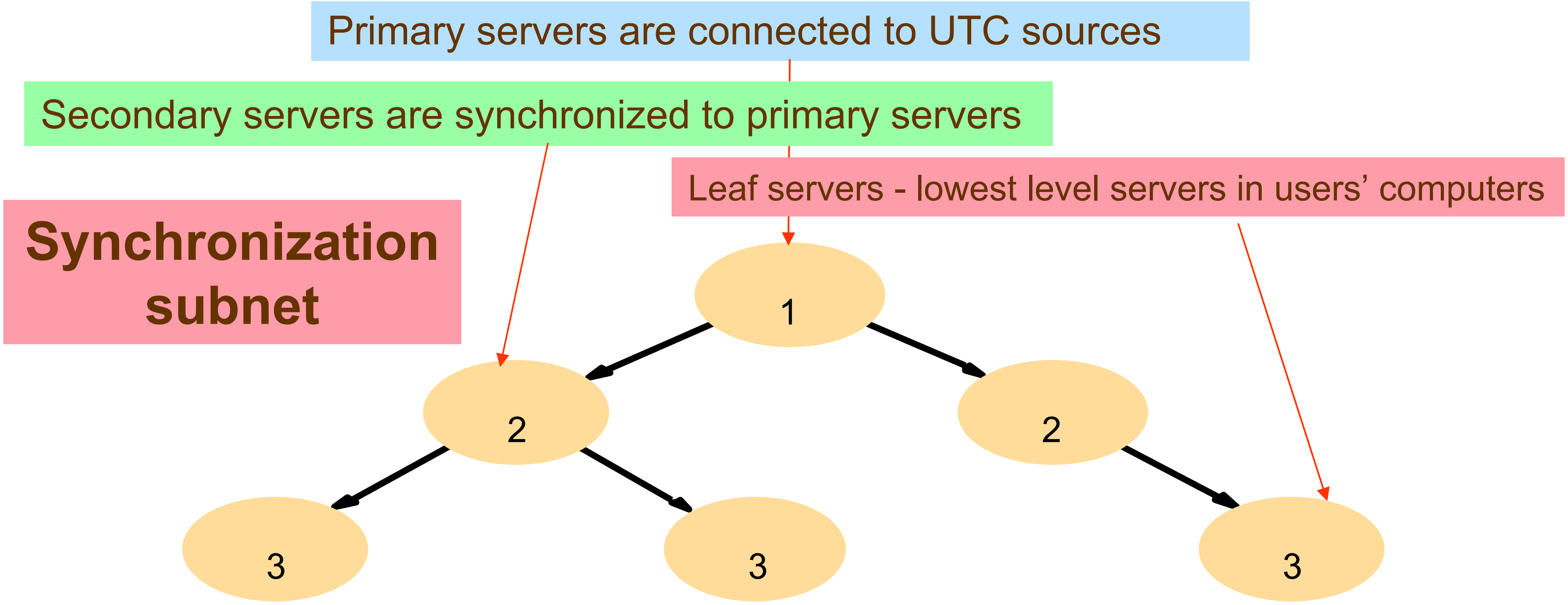
11/11/2005

Berkeley algorithm (skip)

- Cristian's algorithm -
  - a single time server might fail, so they suggest the use of a group of synchronized servers
  - it does not deal with faulty servers
- Berkeley algorithm (also 1989)
  - An algorithm for internal synchronization of a group of computers
  - A *master* polls to collect clock values from the others (*slaves*)
  - The master uses round trip times to estimate the slaves' clock values
  - It takes an average (eliminating any above some average round trip time or with faulty clocks)
  - It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
  - Measurements
    - 15 computers, clock synchronization 20-25 millisecs drift rate < $2x10^{-5}$
    - If master fails, can elect a new master to take over

Network Time Protocol (NTP)

- Cristian’s method and the Berkeley algorithm are intended for intranets
- NTP: a time service for the Internet - synchronizes clients to UTC
  - Reliability from redundant paths, scalable, authenticates time sources
- The synchronization subnet can reconfigure if failures occur, e.g.
  - a primary that loses its UTC source can become a secondary
  - a secondary that loses its primary can use another primary


Primary servers are connected to UTC sources

Secondary servers are synchronized to primary servers

Leaf servers - lowest level servers in users’ computers

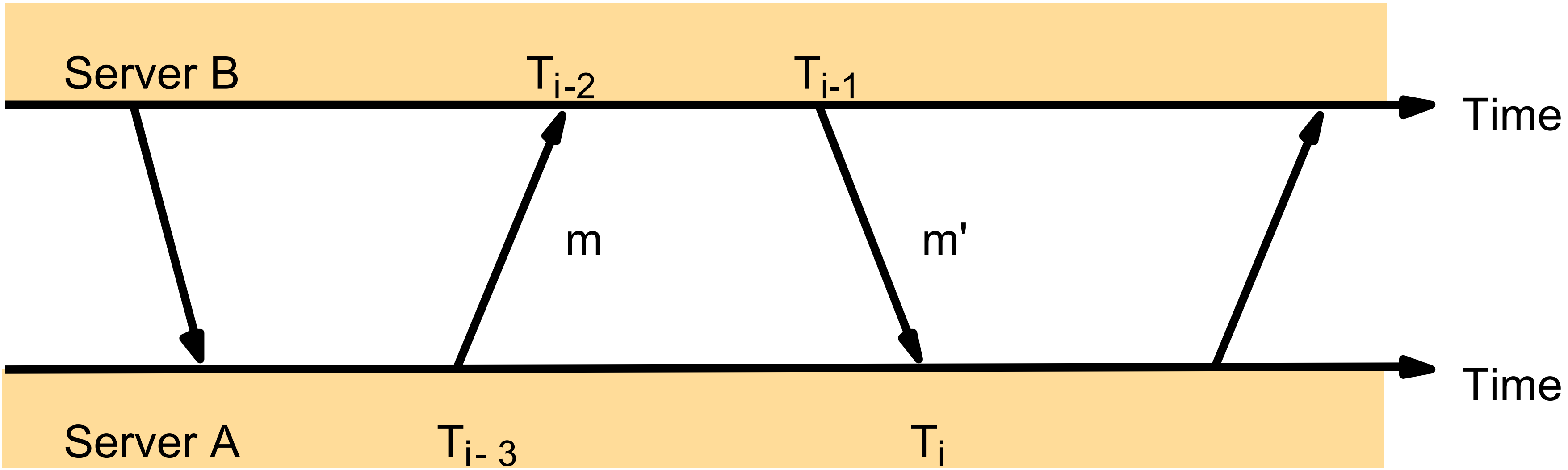
Synchronization subnet

NTP - synchronisation of servers

- 3 Modes of synchronization:
- Multicast
  - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
- Procedure call
  - A server accepts requests from other computers (like Cristiain's algorithm). Higher accuracy. Useful if no hardware multicast.
- Symmetric
  - Pairs of servers exchange messages containing time information
  - Used where very high accuracies are needed (e.g. for higher levels)

Messages exchanged between a pair of NTP peers (skip)

- All modes use UDP
- Each message bears timestamps of recent events:
  - Local times of *Send* and *Receive* of previous message
  - Local times of *Send* of current message
- Recipient notes the time of receipt T_i (we have T_{i-3}, T_{i-2}, T_{i-1}, T_i)
- In symmetric mode there can be a non-negligible delay between messages



11/11/2005

Accuracy of NTP (skip)

- For each pair of messages between two servers, NTP estimates an offset *o*, between the two clocks and a delay d_i (total time for the two messages, which take *t* and *t'*)

  $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$
- This gives us (by adding the equations) :

  $d_i = t + t' = T_{\mathrm{i-2}} - T_{i-3} + T_{\mathrm{i}} - T_{i-1}$
- Also (by subtracting the equations)

  $o = o_{\mathrm{i}} + (t' - t)/2$ where $o_i = (T_{\mathrm{i-2}} - T_{i-3} + T_{i-1} - T_i)/2$
- Using the fact that *t*, *t'*>0 it can be shown that

  $o_i - d_i/2 \leq o \leq o_i + d_i/2$.
  - Thus o_i is an estimate of the offset and d_i is a measure of the accuracy
- NTP servers filter pairs <o_i, d_i>, estimating reliability from variation, allowing them to select peers
- Accuracy of 10s of millisecs over Internet paths (1 on LANs)

Logical time and logical clocks (Lamport 1978)

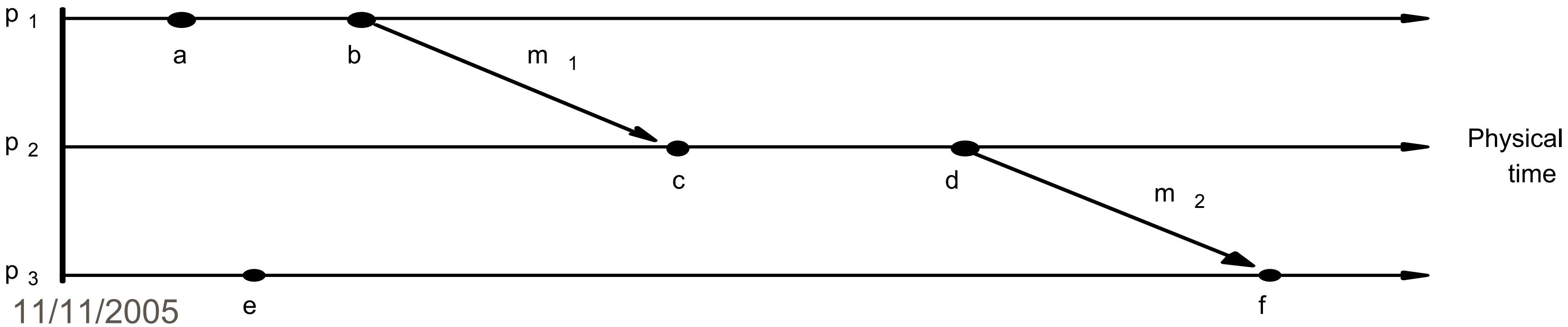
- Instead of synchronizing clocks, event ordering can be used
  - If two events occurred at the same process *pi* (*i* = 1, 2, … *N*) then they occurred in the order observed by *pi*, that is the order $\rightarrow_i$
  - when a message, *m* is sent between two processes, *send(m)* happened before *receive(m)*
- happened-before relation: obtained by generalizing the above two relations
  - denoted by →
  - HB1, HB2 are formal statements of the above two relations
  - HB3 means happened-before is transitive

Not all events are related by →, e.g., a $\nrightarrow$ e and e $\nrightarrow$ a
consider a and e (different processes and no chain of messages to relate them)
they are not related by → ; they are said to be concurrent; write as $a \parallel e$

$a \longrightarrow b$ (at $p1$) $c \longrightarrow d$ (at $p2$)

$b \longrightarrow c$ because of $m1$

also $d \longrightarrow f$ because of $m2$



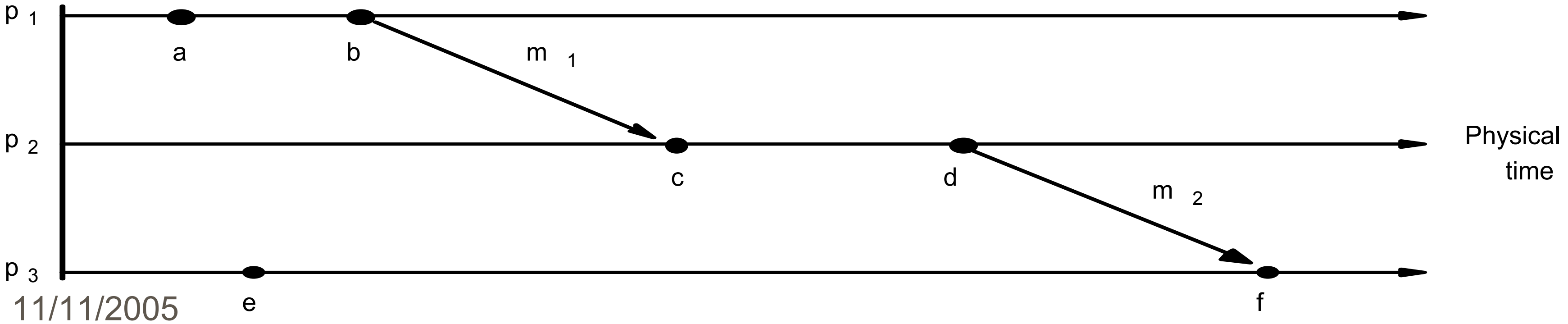
Lamport's logical clocks

- A logical clock is a monotonically increasing software counter. It need not relate to a physical clock
- Each process p_i has a logical clock, L_i which can be used to apply logical timestamps to events
  - LC1: L_i is incremented by 1 before each event at process p_i
  - LC2: (a) when process p_i sends message *m*, it piggybacks $t = L_i$
    (b) when p_j receives *(m,t)*, it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive* (*m*)
- e → e' ⇒ L(e) < L(e') but not vice versa, example?

L(*e*) < *L*(b) but *e* || b

each of p1, p2, p3 has its logical clock initialised to zero, the clock values are those immediately after the event. e.g. 1 for a, 2 for b.

for *m*1, 2 is piggybacked and *c* gets *max*(0,2)+1 = 3



Vector clocks (Mattern [1989] and Fidge [1991])

- Vector clocks overcome the shortcoming of Lamport logical clocks
  - (L(e) < L(e') does not imply e happened before e')

- Vector clock V_i at process p_i is an array of N integers, a vector
- Each process keeps its own vector clock V_i ,used to timestamp local events
- *$V_i[i]$ is the number of events that p_i has timestamped*
- *$V_i[j]$ ($j \neq i$) is the number of events at p_j that p_i has been affected by*

Rules for updating clocks:

- VC1:initially $V_i[j] = 0$ for $i, j = 1, 2, \ldots N$
- VC2:before p_i timestamps an event it sets $V_i[i] := V_i[i] + 1$
- VC3:p_i piggybacks $t = V_i$ on every message it sends
- VC4:when p_i receives (m,t) it sets $V_i[j] := \max(V_i[j] , t[j])$ $j = 1, 2, \ldots N$ *(then before next event adds I to own element using VC2)*
  - *Merge operation*

- *E.g. at p_2, (0, 0, 0) -> (0, 1, 0) -> (0, 2, 0) -> (0, 3, 0) … -> (1, 4, 3)*
- *Now, received a mes. from p_3 that piggybacks t = (1, 0, 3),*

Compare vector timestams

- Meaning of =, <=, < for vector timestamps - compare elements pairwise

(1) $V = V'$ iff $V[j] = V'[j]$ for $j = 1, 2, \ldots, N$

(2) $V \leq V'$ iff $V[j] \leq V'[j]$ for $j = 1, 2, \ldots, N$

(3) $V < V'$ iff $V \leq V'$ and $V \neq V'$

Examples:

V_1	and	V_2
(1, 3, 2)		(1, 3, 3)
(1, 3, 2)		(1, 3, 0)
(1, 3, 2)		(1, 3, 2)
(1, 3, 2)		(2, 3, 1)

Vector clock example

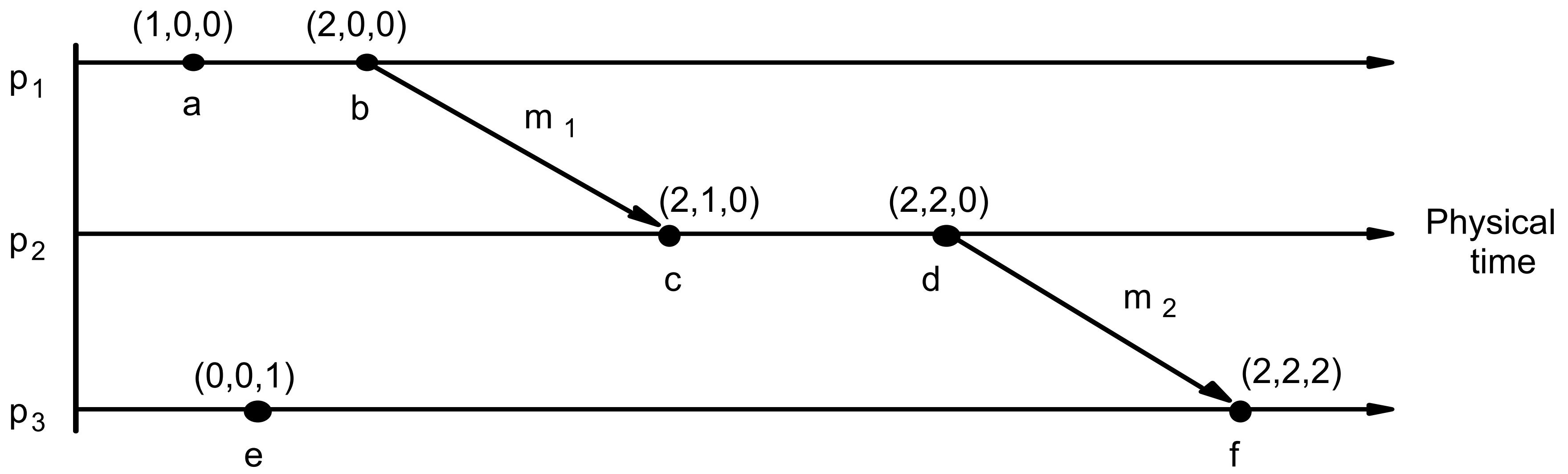
At p_1: a(1,0,0) b (2,0,0) piggyback (2,0,0) on m_1

At p_2: on receipt of m_1 get *max* ((0,0,0), (2,0,0)) = (2, 0, 0) add 1 to own element = (2,1,0)

- Note that $e \rightarrow e'$ implies $V(e) < V(e')$. The converse is also true. (assignment)
- V(a) < V(f)

Can you see a pair of parallel events?

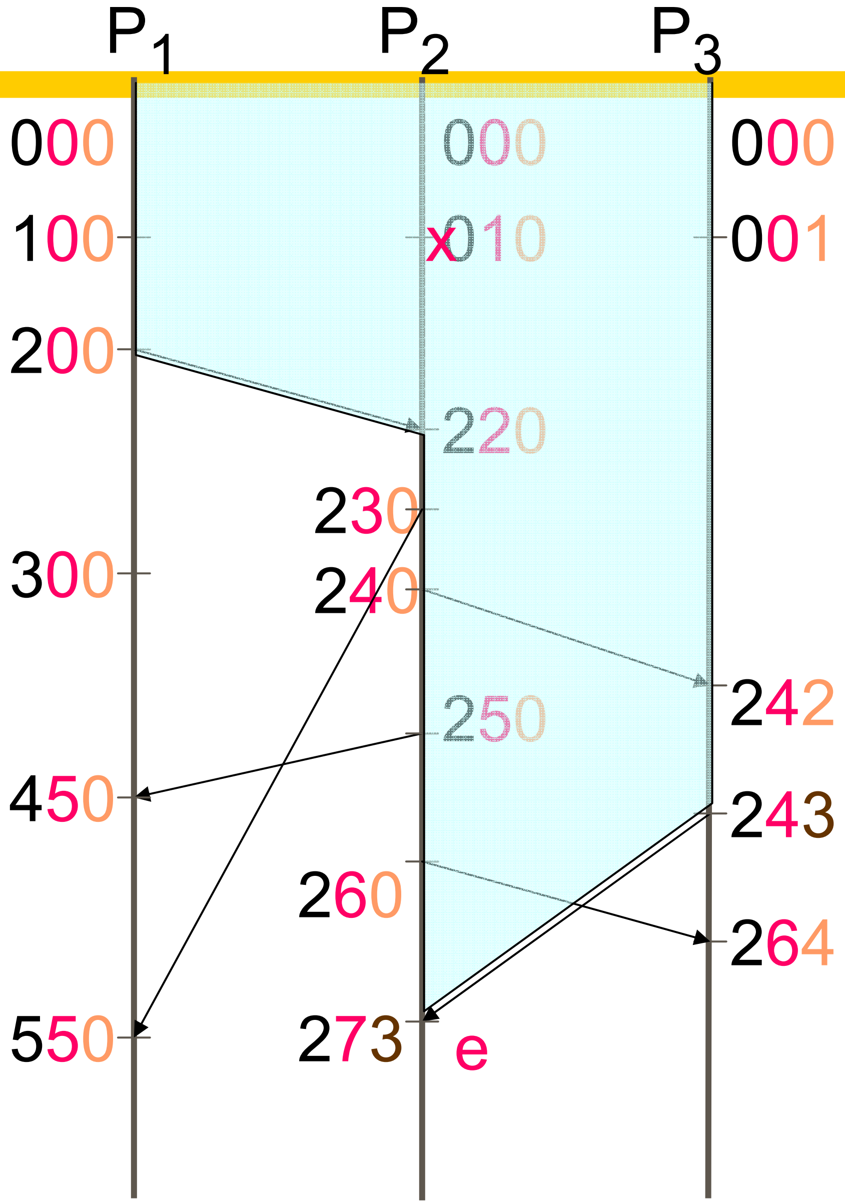
c || *e*(parallel) because neither *V*(c) <= *V*(*e*) nor *V*(*e*) <= *V*(*c*).



Vector clock example

For fixed event e.

x → e iff
V(x) < V(e)

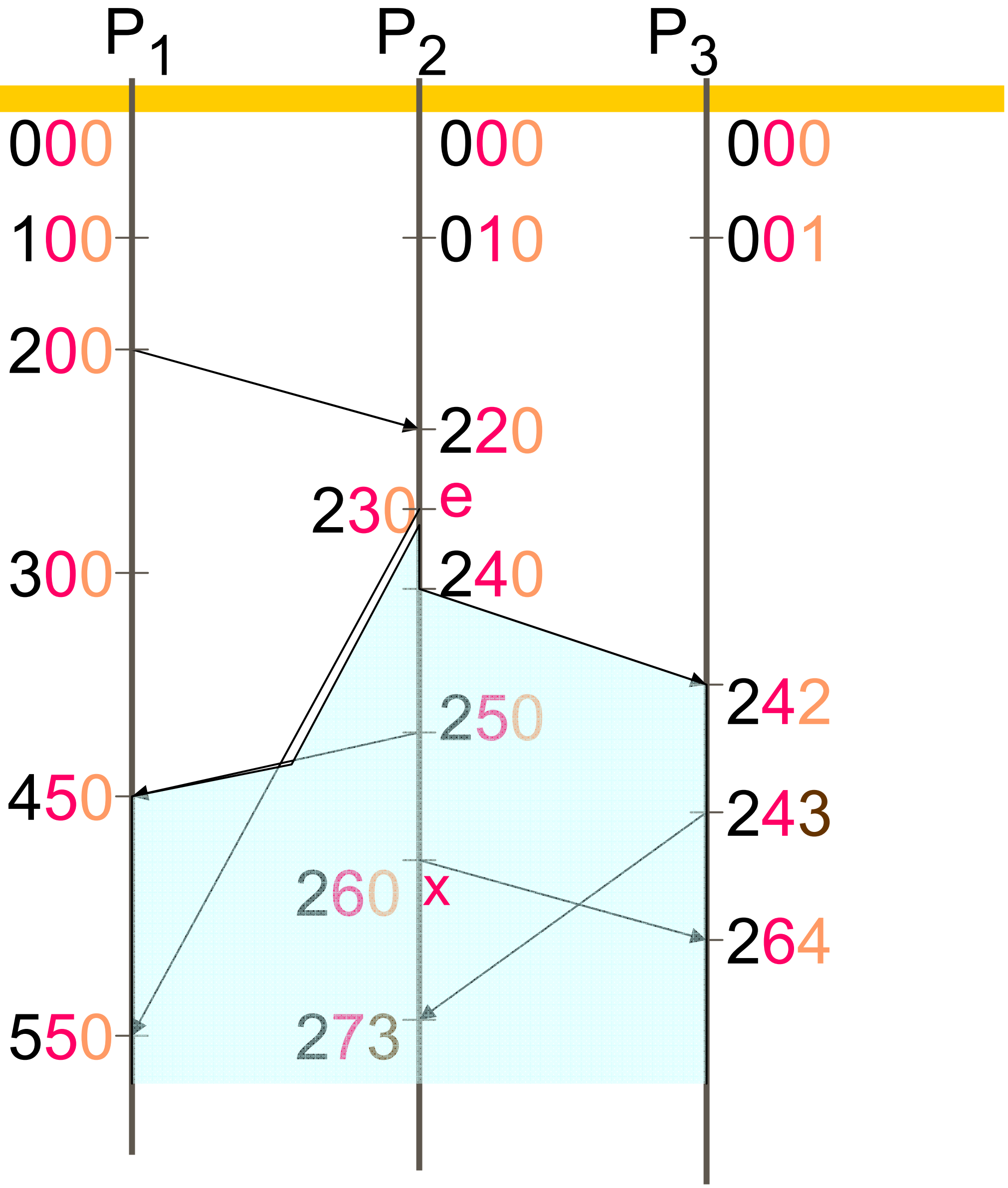


11/11/2005

Vector clock example

For fixed event e.

e → x iff
V(e) < V(x)

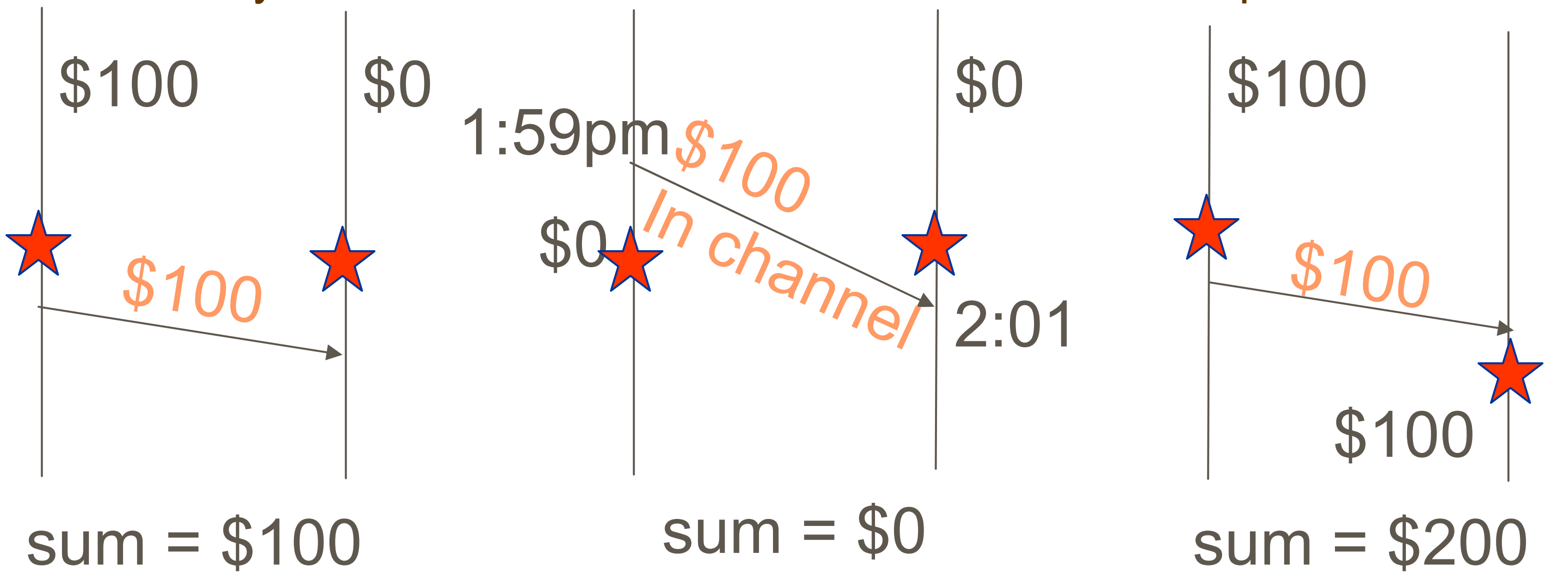


Summary on time and clocks in distributed systems

- accurate timekeeping is important for distributed systems.
- algorithms (e.g. Cristian's and NTP) synchronize clocks in spite of their drift and the variability of message delays.
- for ordering of an arbitrary pair of events at different computers, clock synchronization is not always practical.
- the happened-before relation is a partial order on events that reflects a flow of information between them.
- Lamport clocks are counters that are updated according to the happened-before relationship between events.
- vector clocks are an improvement on Lamport clocks,
  - we can tell whether two events are ordered by happened-before or are concurrent by comparing their vector timestamps

11.5 Global states

- We are interested in a consistent global state. Intuitively, it means a set of process states + channel states.
- As in transaction systems, it is sometimes desirable to store checkpoints of a distributed system to be able to restart from a well-defined past state after a crash.



(a)

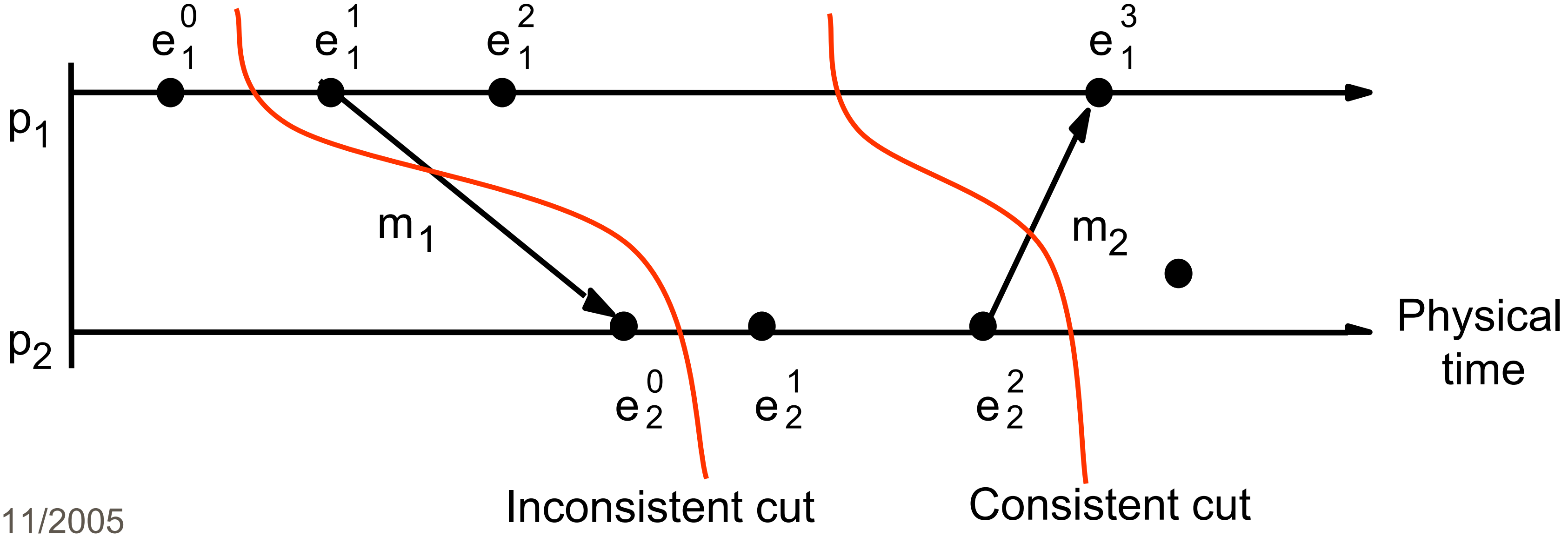
(b)
message delay

(c)
not synchronized

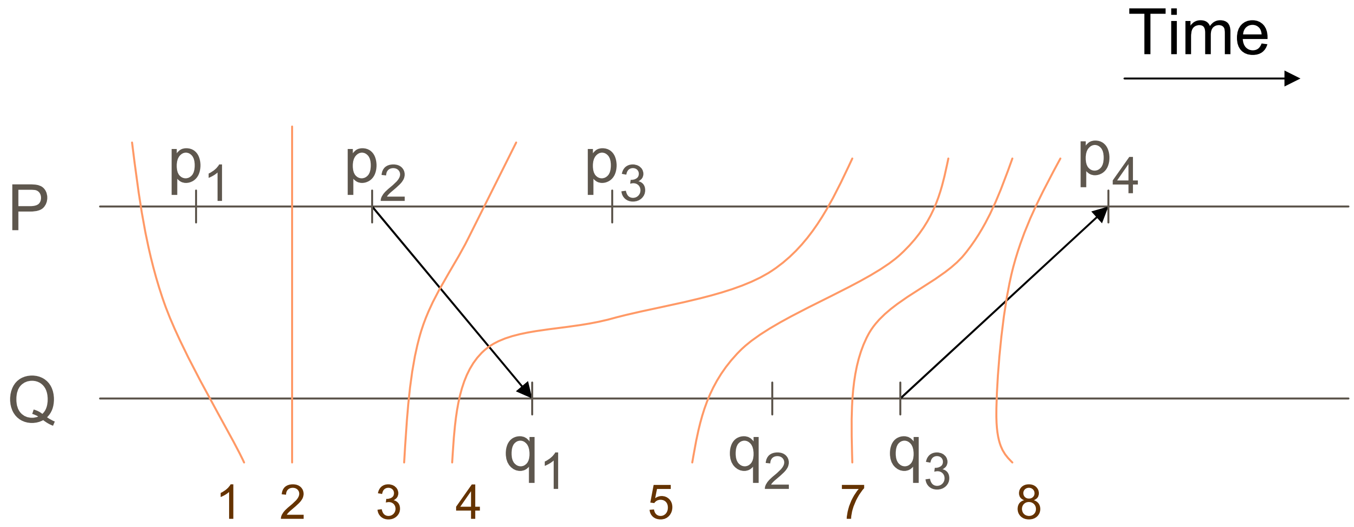
Snapshots taken at 2:00pm by local clocks ★

Cuts

- A cut C can be represented by a curve in the time-process diagram which crosses all process lines.
- C divides all events to P_C (those happened before C) and F_C (future events)
- Cut C is consistent if there is no message whose sending event is in F_C and whose receiving event is in P_C
- Cuts are made on states. A cut corresponds to the set of states it crosses.
  - A consistent (inconsistent) cut leads to a consistent (inconsistent) set of states.



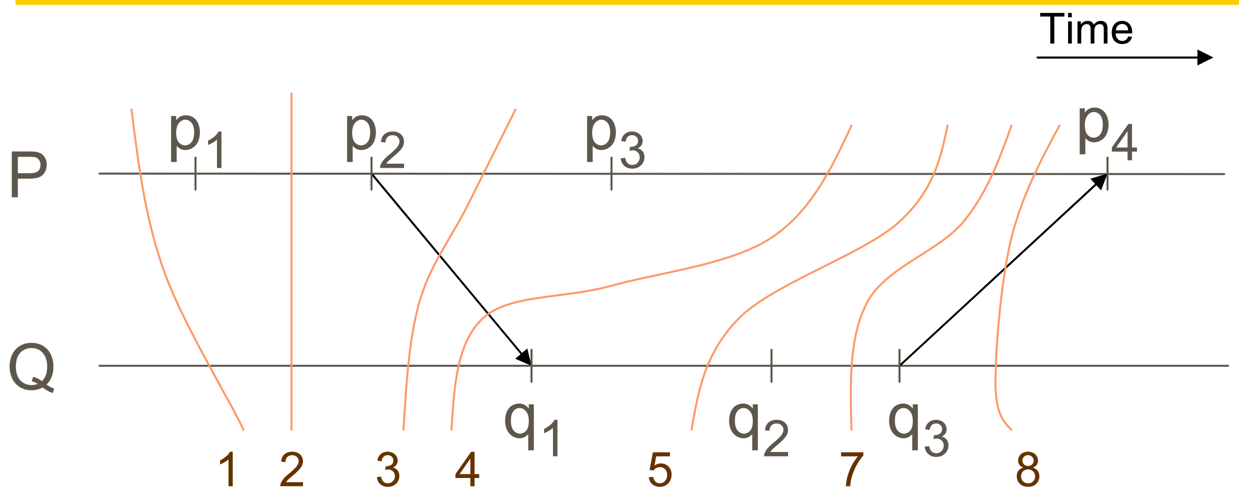
Progress shown by cuts



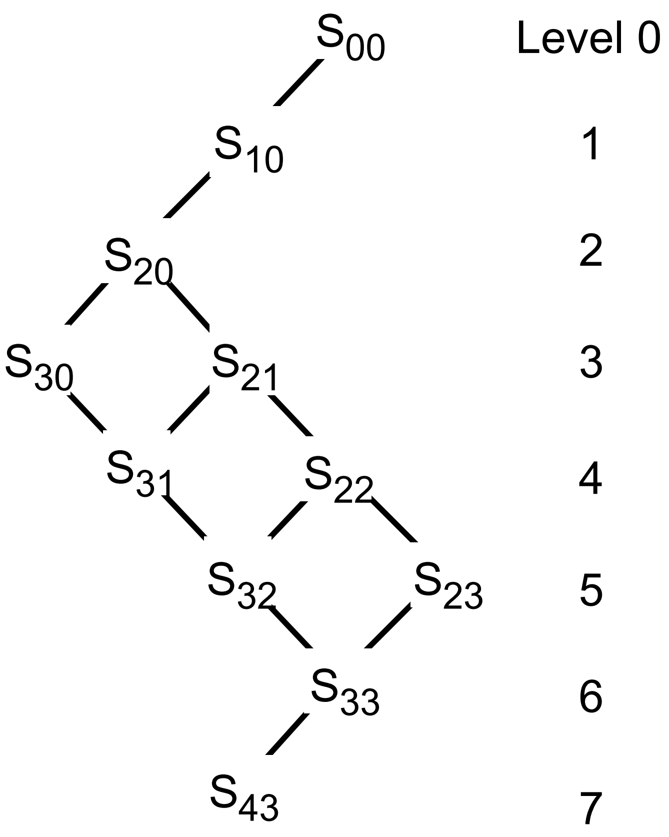
One ordering of a series of consistent global states (cuts), corresponding to one possible actual flow of the global states of the DS. It can be derived from the partial order of vector clocks, which contains all the possible flows.

How many possible cuts are there? 5*4 = 20

The lattice of global states

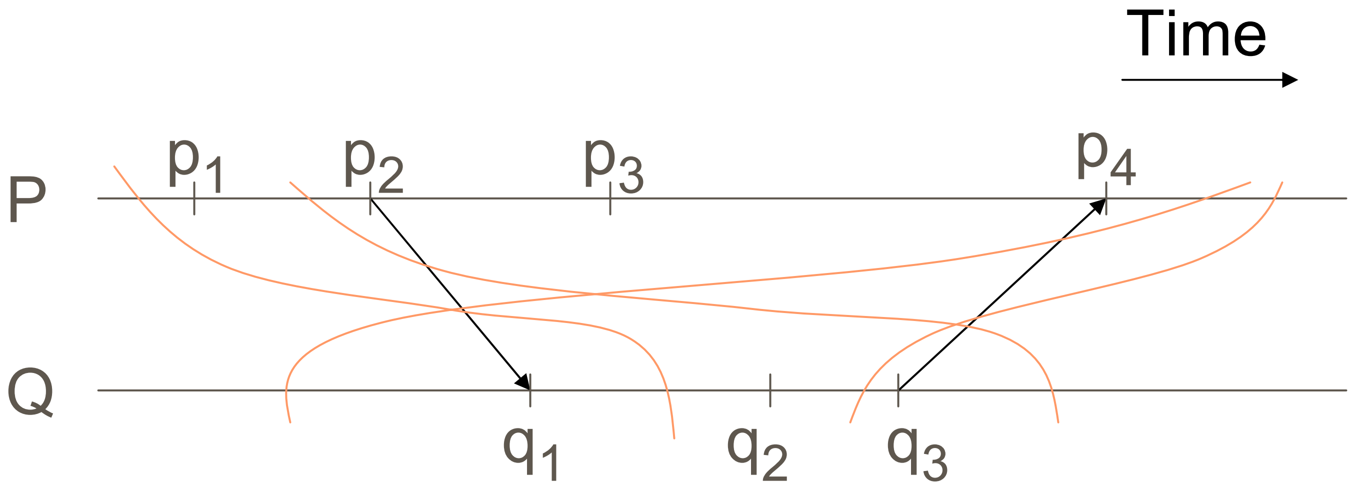


S_{ij} = global state after *i* events at P and *j* events at Q



Lattice represents partial order.
All consistent global states can be put in the "lattice of global states"
And, all possible flows can be derived from the lattice, the one in the above figure is only one of them

Inconsistent cuts



How many inconsistent cuts are there?

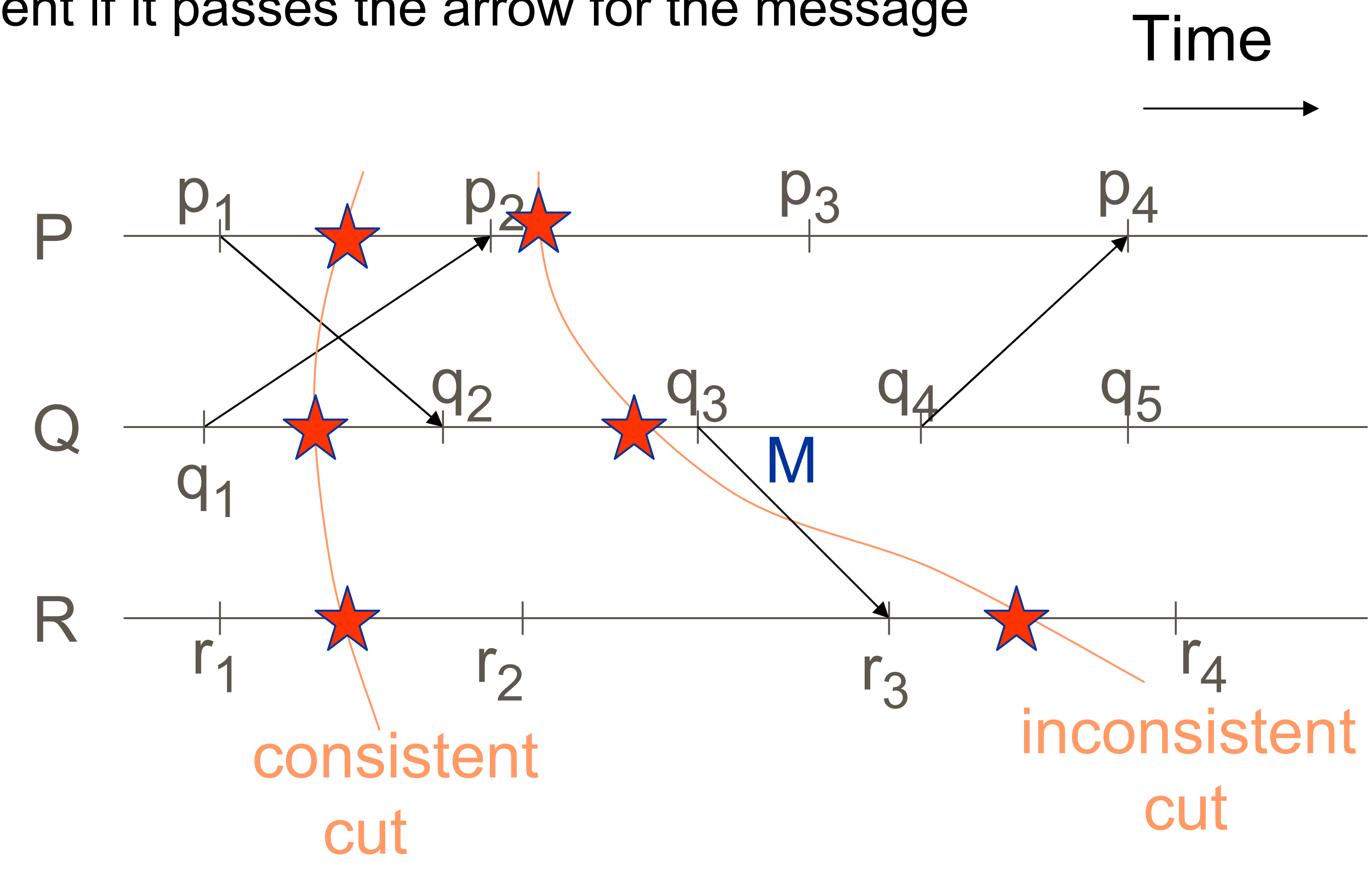
2*3 + 1*3 = 9 are inconsistent, and 11 are consistent.

Inconsistent cut cannot actually happen →States in Inconsistent cut could not have coexisted.

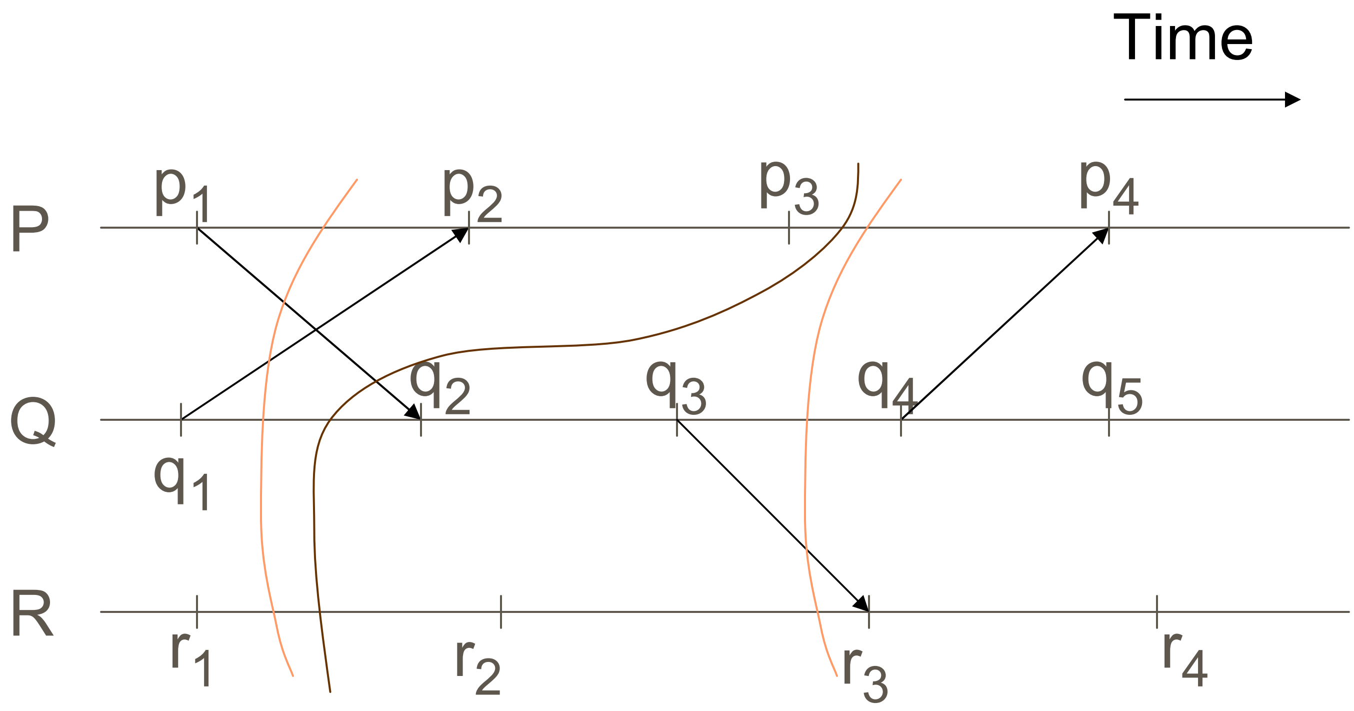
More examples

Can we derive a “cheap” rule for making quick decision?

A cut is inconsistent if it passes the arrow for the message that it crosses



More consistent cuts



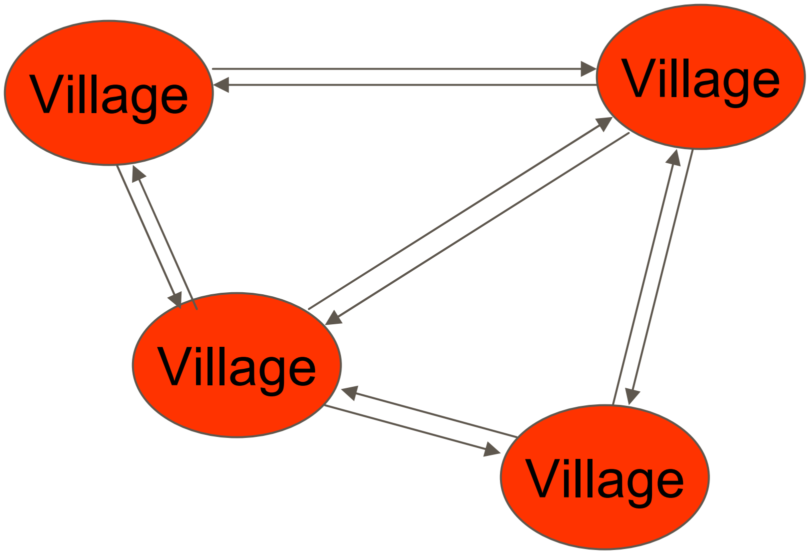
Apply the “cheap” rule to verify

Checkpointing

- Cut C is consistent → C doesn’t contradict sequence of events experienced by any site → can assume it did exist at the same time
- Can use snapshot as checkpoint, from which activity in distributed system can be resumed after crash

SNAPSHOT algorithm analog: census taking

- Chandy and Lamport [1985] describe a SNAPSHOT algorithm for determining global states of DS. The goal is to record a set of process and channel states (a snapshot) for a set of processes p_i ($i = 1, 2, \ldots, N$)
- "Census taking in ancient kingdom": want to take census counting all people, some of whom may be traveling on highways



FIFO road

villages are strongly connected

Census taking algorithm

- Close all gates into/out of each village (process) and count people (record process state) in village; these actions need not be synched with other villages
- Open each outgoing gate and send official with a red cap (special marker message).
- Open each incoming gate and count all travelers (record channel state = messages sent but not received yet) who arrive ahead of official.
- Tally the counts from all villages.

- In fact, it works as long as at least one village initiates census taking.
- The termination condition is, each village sees the arrival of a red-capped official on every incoming road.
- Note that at termination, every road has been traversed by an official exactly once

Algorithm SNAPSHOT

- All processes are initially white: Messages sent by white(red) processes are also white (red)
- MSend [Marker sending rule for process P]
  - Suspend all other activities until done
  - Record P's state
  - Turn red
  - Send one marker over each output channel of P.
- MReceive [Marker receiving rule for P]

  On receiving marker over channel C,
  - if P is white { Record state of channel C as empty;

    Invoke MSend; }
  - else record the state of C as sequence of white messages received since P turned red.
  - Stop when marker is received on each incoming channel

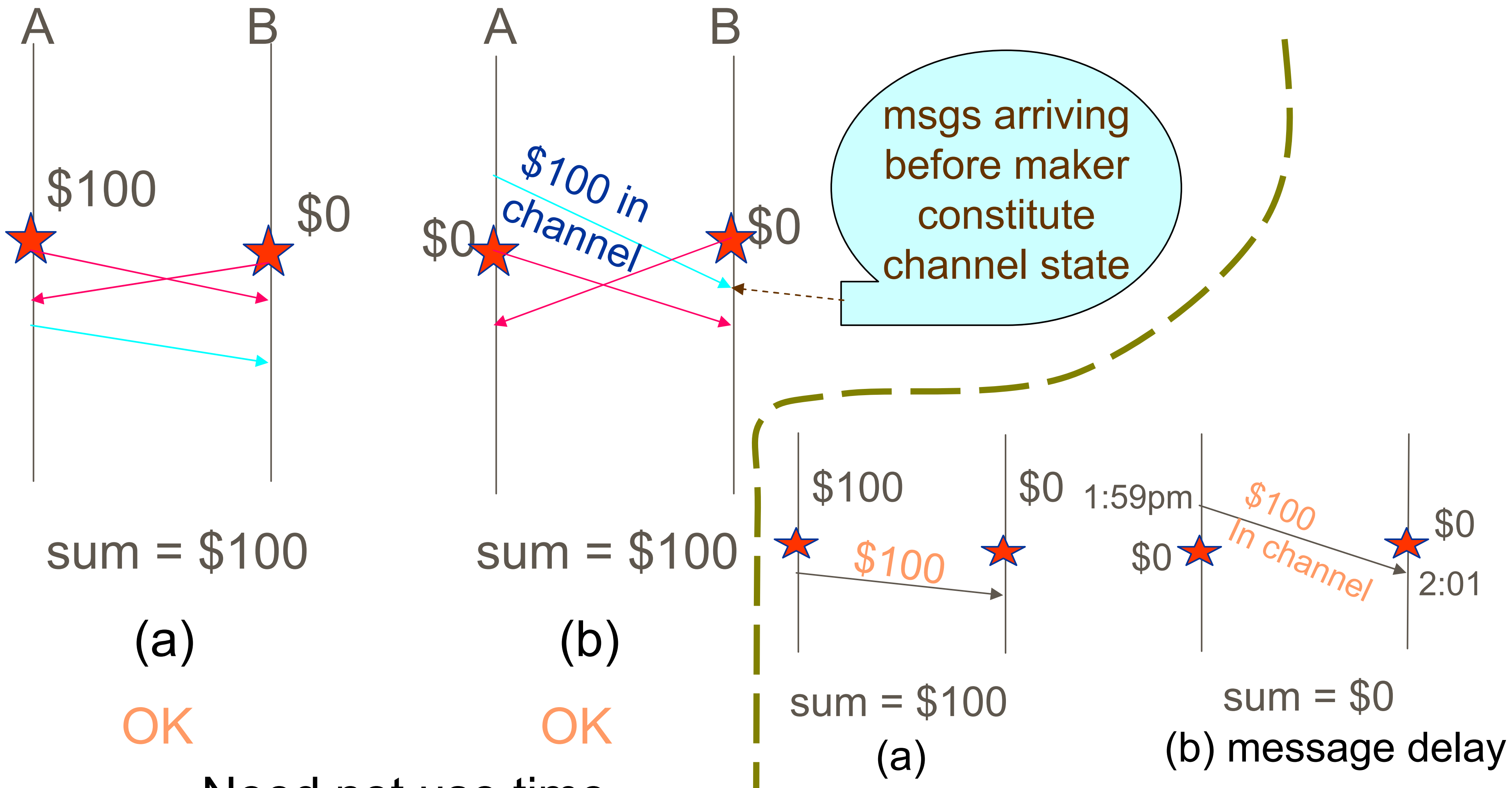
Property of SNAPSHOT

- If network is strongly connected and at least one process initiates MSend, then SNAPSHOT

  will take consistent global snapshot (collection of process states and channel states).
- i.e. SNAPSHOT makes consistent cuts

- The processes may continue their execution and send and receive normal messages while the snapshot takes place

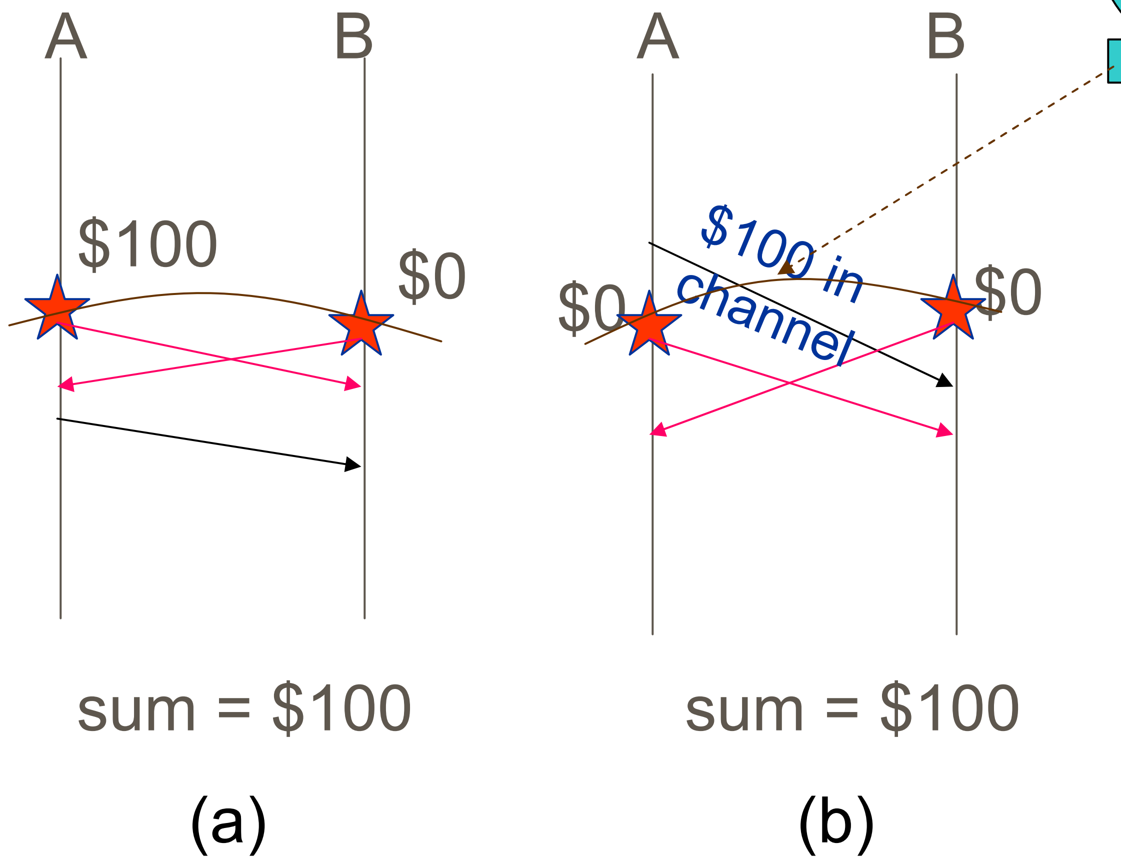
Snapshots taken by SNAPSHOT algorithm



msgs arriving before maker constitute channel state

Need not use time.

Cuts corresponding to snapshots

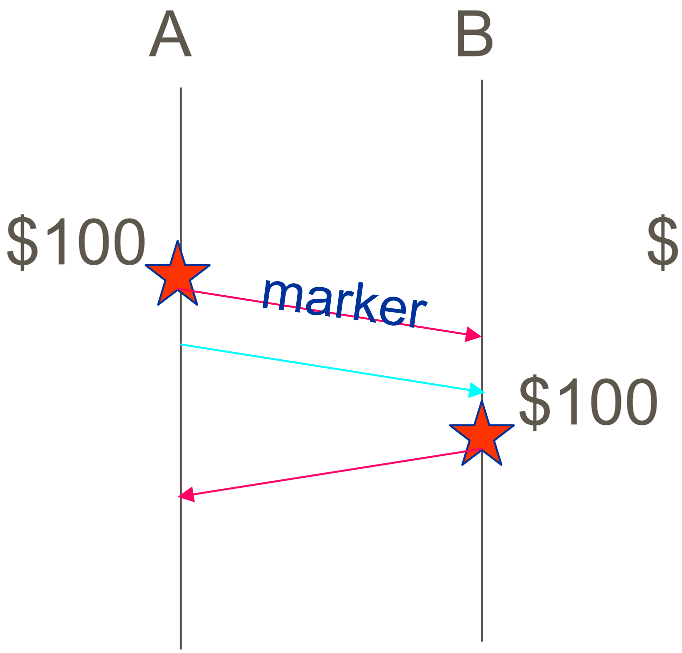


Note that they intersect

Snapshot only generates consistent cuts

11/11/2005

Snapshots taken by SNAPSHOT



A

B

$100

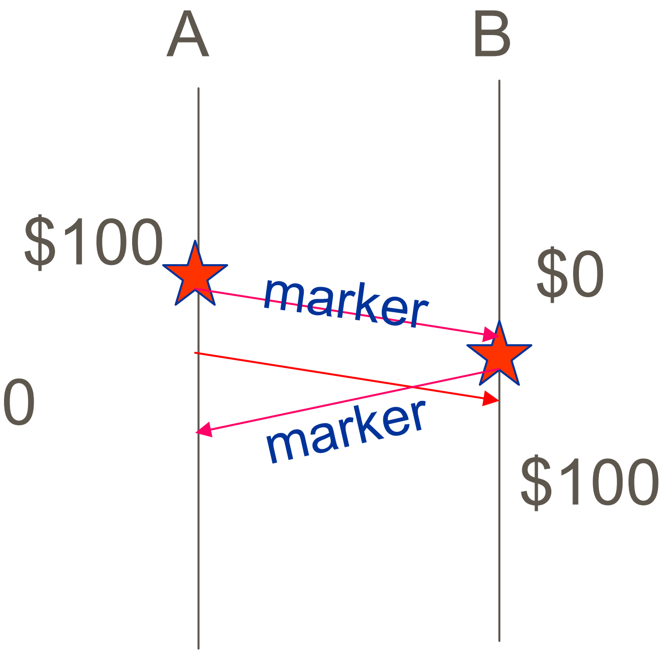
marker

$100

sum = $200

(c)

Cannot happen



A

B

$100

marker

$0

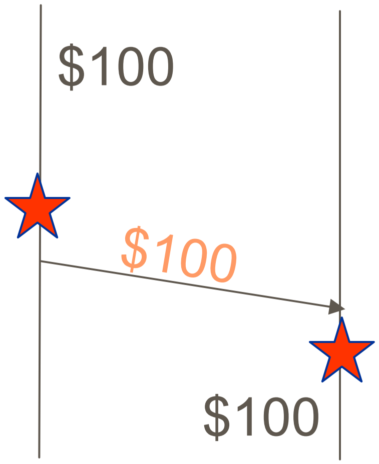
marker

$100

sum = $100

(c')

Will be like this



$100

$100

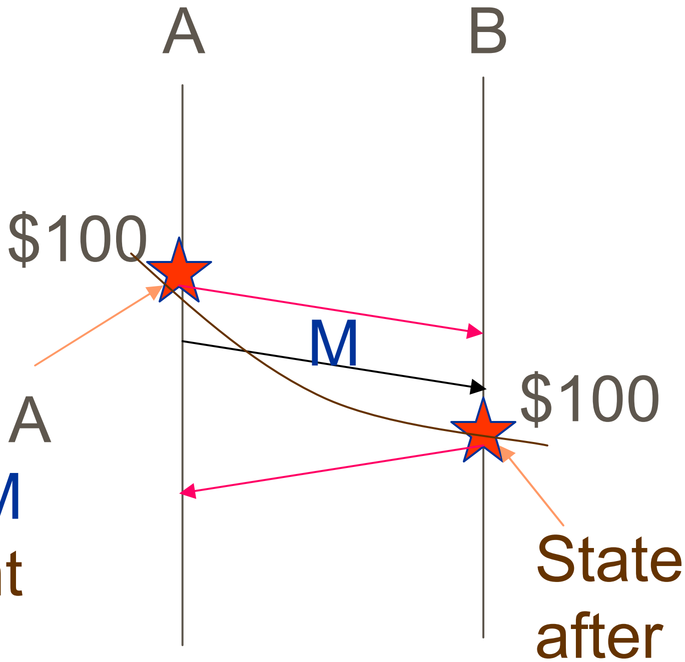
$100

sum = $200

(c) not synchronized

11/11/2005

Cut corresponding to snapshot

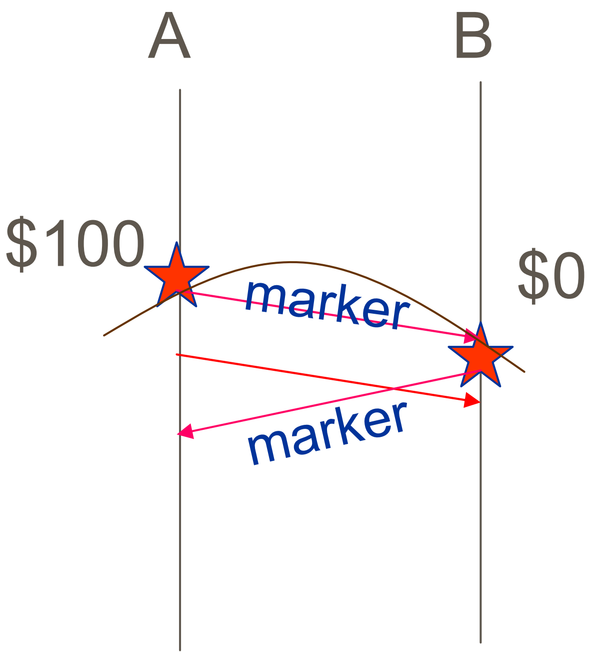


State of A
before M
was sent

State of B
after M
was received

sum = $200

Msg M goes from future to past →
SNOPSHOP never generates such cut



sum = $100

(c’)

Will be like this