Logical Time and Clocks

Ken Birman

Cornell University. CS5410 Fall 2008.

Recall cloud "layers"

- Highest level consists of applications
- These are composed from services that run on data harvested by applications using tools Map-Reduce
- The overall system is managed by a collection of core infrastructure services, such as locking and node status tracking
- How can we "reason" about the behavior of such components?
  - The scale and complexity makes it seem hard to say more than "Here's a service. This is what it does"

But we can do more

- We can describe distributed systems in more rigorous ways that let us say stronger things about them
- The trick is to start at the bottom, not the top
- This week: we'll focus on concepts of *time* as they arise in distributed systems

What time is it?

- In distributed system we need practical ways to deal with time
  - E.g. we may need to agree that update A occurred before update B
  - Or offer a "lease" on a resource that expires at time 10:10.0150
  - Or *guarantee* that a time critical event will reach all interested parties within 100ms

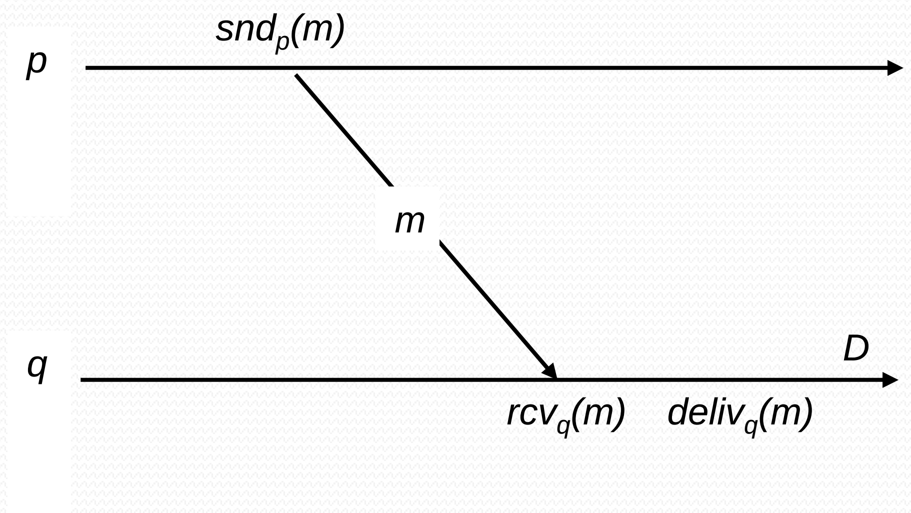
But what does time “mean”?

- Time on a global clock?
  - E.g. with GPS receiver
- … or on a machine’s local clock
  - But was it set accurately?
  - And could it drift, e.g. run fast or slow?
  - What about faults, like stuck bits?
- … or could try to agree on time

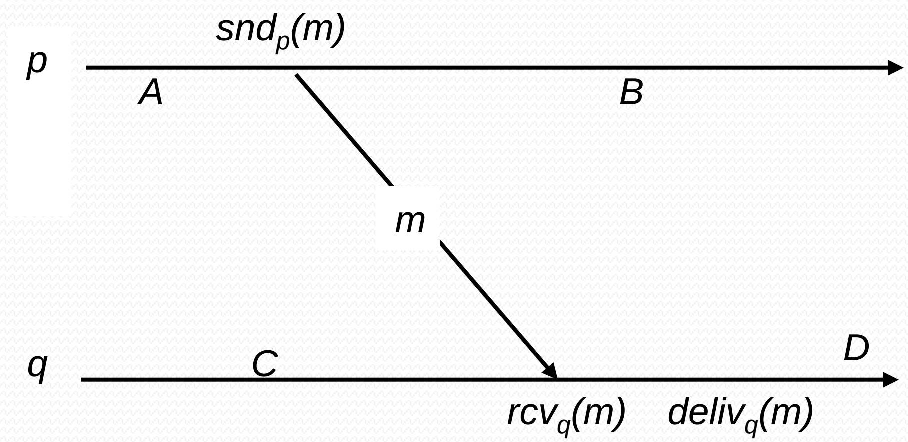
Lamport's approach

- Leslie Lamport suggested that we should reduce time to its basics
  - Time lets a system ask "Which came first: event A or event B?"
  - In effect: time is a means of labeling events so that…
    - If A happened before B, TIME(A) < TIME(B)
    - If TIME(A) < TIME(B), A happened before B

Drawing time-line pictures:

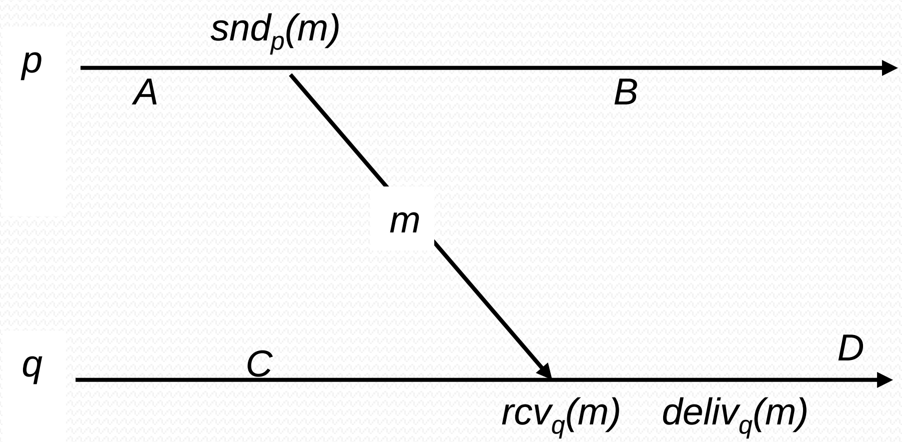


Drawing time-line pictures:



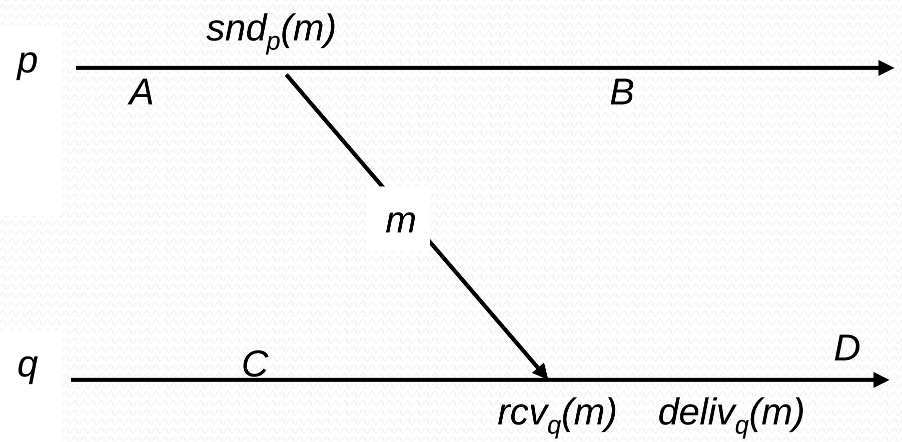
- A, B, C and D are “events”.
  - Could be anything meaningful to the application
  - So are snd(m) and rcv(m) and deliv(m)
- What ordering claims are meaningful?

Drawing time-line pictures:



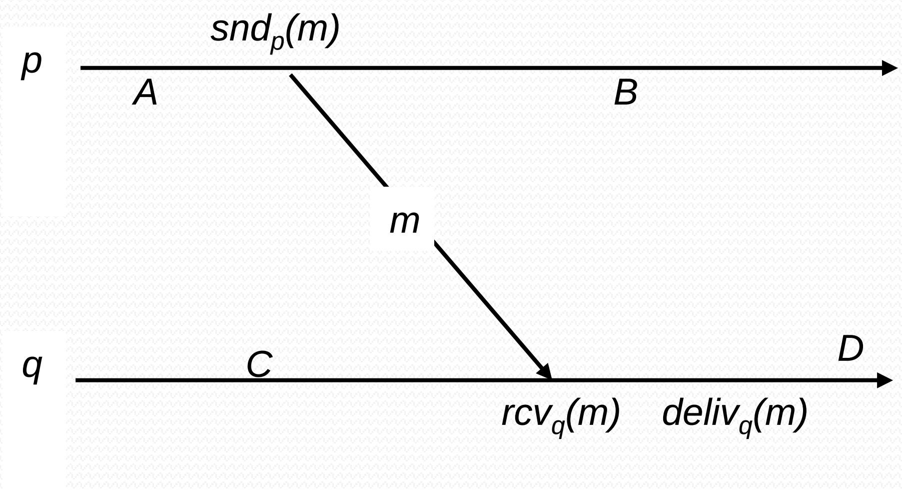
- A happens before B, and C before D
  - "Local ordering" at a single process
  - Write $A \xrightarrow{p} B$ and $C \xrightarrow{q} D$

Drawing time-line pictures:



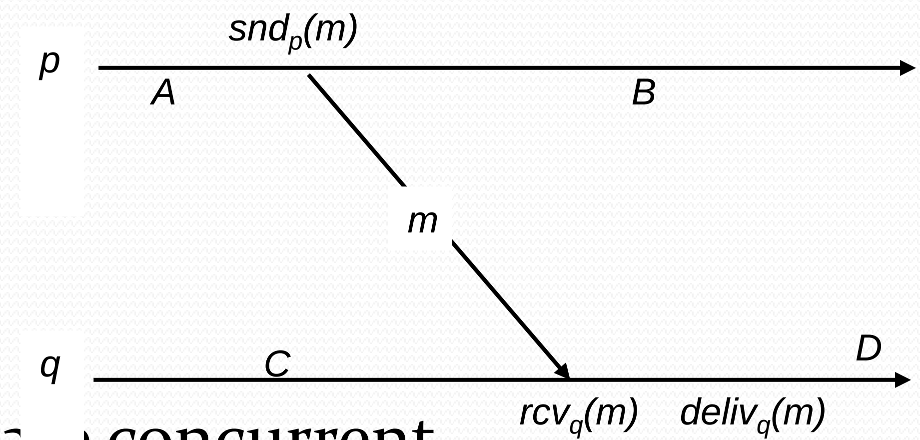
- $snd_p(m)$ also happens before $rcv_q(m)$
  - "Distributed ordering" introduced by a message
  - Write $snd_p(m) \xrightarrow{M} rcv_q(m)$

Drawing time-line pictures:



- A happens before D
  - Transitivity: A happens before $snd_p(m)$, which happens before $rcv_q(m)$, which happens before D

Drawing time-line pictures:



- B and D are concurrent
  - Looks like B happens first, but D has no way to know. No information flowed…

Happens before "relation"

- We'll say that "A happens before B", written $A \rightarrow B$, if
  1. $A \rightarrow^{P} B$ according to the local ordering, or
  2. A is a *snd* and B is a *rcv* and $A \rightarrow^{M} B$*, or*
  3. A and B are related under the transitive closure of rules (1) and (2)
- So far, this is just a mathematical notation, not a "systems tool"

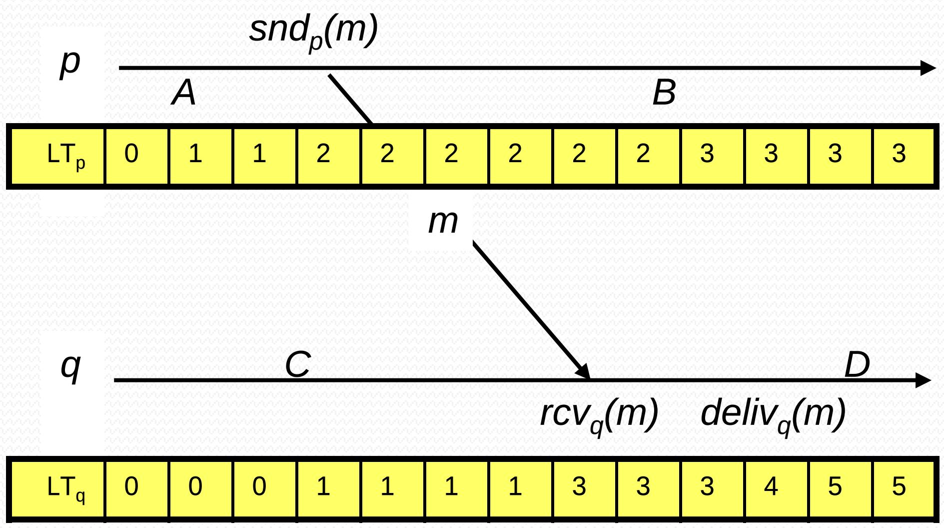
Logical clocks

- A simple tool that can capture parts of the happens before relation
- First version: uses just a single integer
  - Designed for big (64-bit or more) counters
  - Each process *p* maintains LT_p, a local counter
  - A message *m* will carry LT_m

Rules for managing logical clocks

- When an event happens at a process *p* it increments LT_p.
  - Any event that matters to *p*
  - Normally, also *snd* and *rcv* events (since we want receive to occur “after” the matching send)
- When p sends *m*, set
  - $LT_m = LT_p$
- When q receives *m*, set
  - $LT_q = \max(LT_q, LT_m)+1$

Time-line with LT annotations



- LT(A) = 1, LT(snd_p(m)) = 2, LT(m) = 2
- LT(rcv_q(m))=max(1,2)+1=3, etc…

Logical clocks

- If A happens before B, $A \rightarrow B$, then $LT(A) < LT(B)$
- But converse might not be true:
  - If $LT(A) < LT(B)$ can't be sure that $A \rightarrow B$
  - This is because processes that don't communicate still assign timestamps and hence events will "seem" to have an order

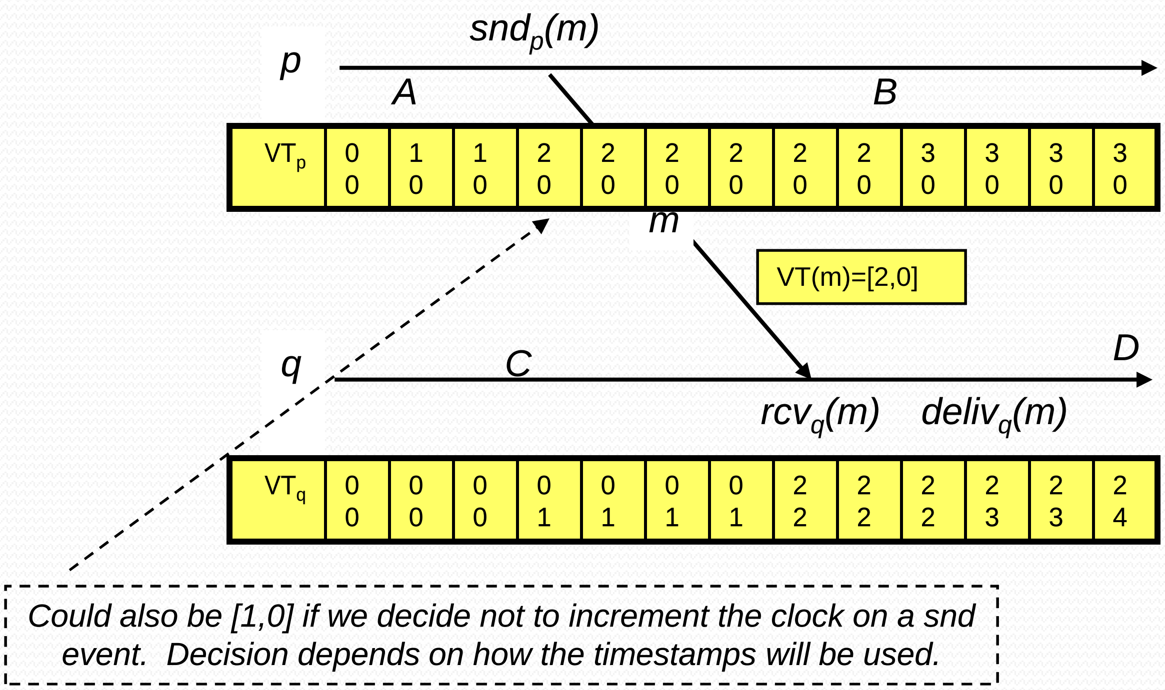
Can we do better?

- One option is to use *vector* clocks
- Here we treat timestamps as a list
  - One counter for each process
- Rules for managing vector times differ from what did with logical clocks

Vector clocks

- Clock is a vector: e.g. VT(A)=[1, 0]
  - We'll just assign p index 0 and q index 1
  - Vector clocks require either agreement on the numbering, or that the actual process id's be included with the vector
- Rules for managing vector clock
  - When event happens at p, increment $VT_p[index_p]$
    - Normally, also increment for snd and rcv events
  - When sending a message, set $VT(m)=VT_p$
  - When receiving, set $VT_q=max(VT_q, VT(m))$

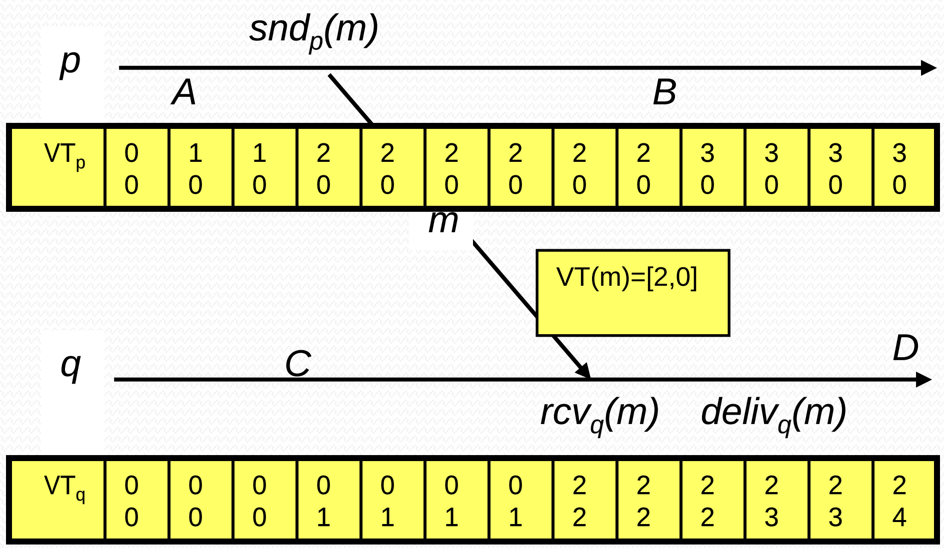
Time-line with VT annotations



Rules for comparison of VTs

- We'll say that $VT_A \leq VT_B$ if
  - $\forall_I$, $VT_A[i] \leq VT_B[i]$
- And we'll say that $VT_A < VT_B$ if
  - $VT_A \leq VT_B$ but $VT_A \neq VT_B$
  - That is, for some i, $VT_A[i] < VT_B[i]$
- Examples?
  - [2,4] ≤ [2,4]
  - [1,3] < [7,3]
  - [1,3] is "incomparable" to [3,1]

Time-line with VT annotations



- VT(A)=[1,0]. VT(D)=[2,4]. So VT(A)<VT(D)
- VT(B)=[3,0]. So VT(B) and VT(D) are incomparable

Vector time and happens before

- If $A \rightarrow B$, then $VT(A) < VT(B)$
  - Write a chain of events from A to B
  - Step by step the vector clocks get larger
- If $VT(A) < VT(B)$ then $A \rightarrow B$
  - Two cases: if A and B both happen at same process p, trivial
  - If A happens at p and B at q, can trace the path back by which q “learned” $VT_A[p]$
- Otherwise A and B happened concurrently

Temporal snapshots

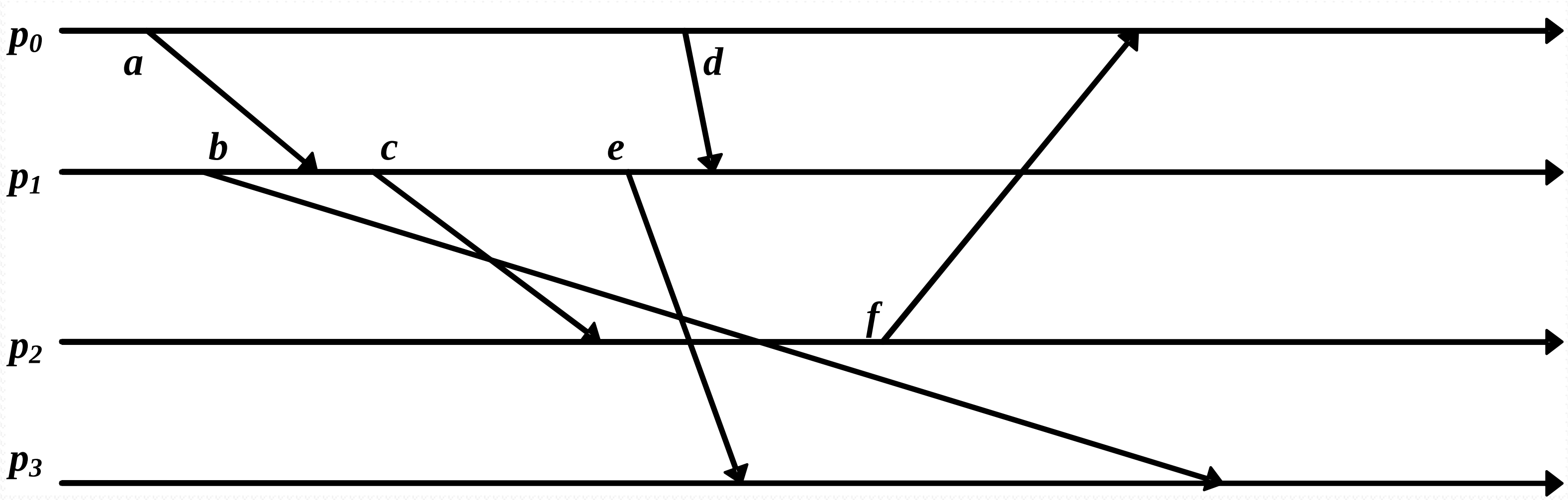
- Suppose that we want to take a photograph of a system while it executes: our goal is to capture the state of each node and each channel at some instant in time
- We can see now that the notion of an "instant in time" is tricky
  - For example, if each node writes down its state at logical time 10000, would this be a "snapshot" that corresponds to anything an external user would perceive as "time"?
  - …. Clearly not. My logical clock could advance much faster than yours

Temporal distortions

- Things can be complicated because we can’t predict
  - Message delays (they vary constantly)
  - Execution speeds (often a process shares a machine with many other tasks)
  - Timing of external events
- Lamport looked at this question too

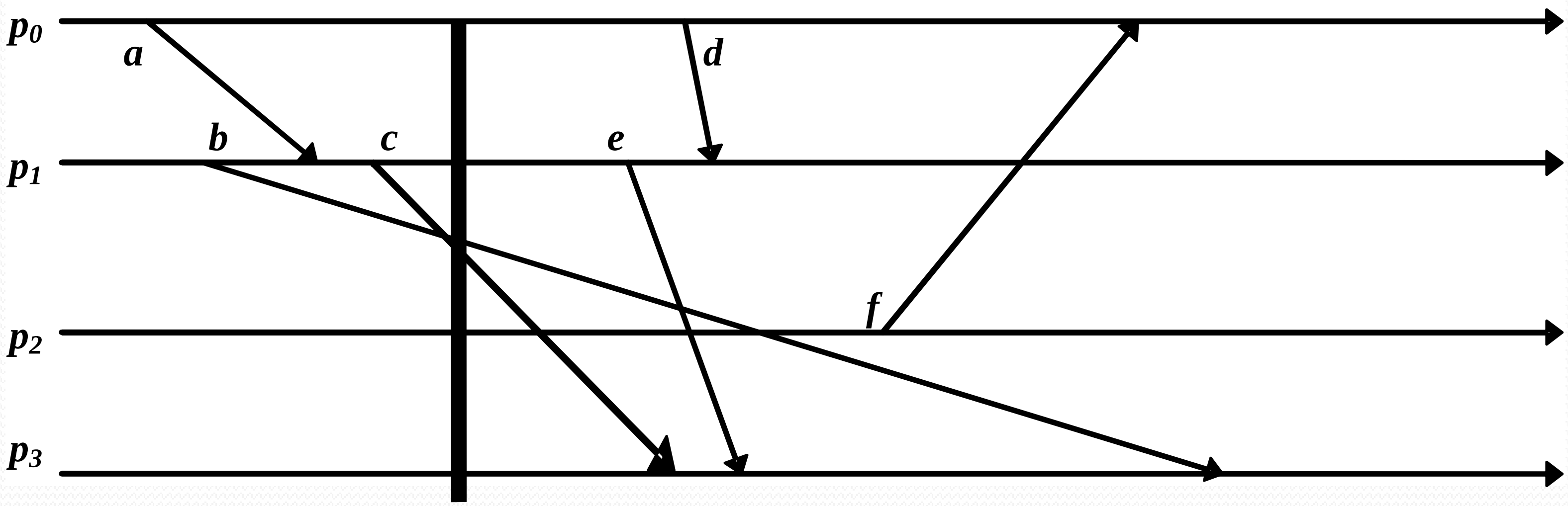
Temporal distortions

- What does "now" mean?



Temporal distortions

- What does “now” mean?



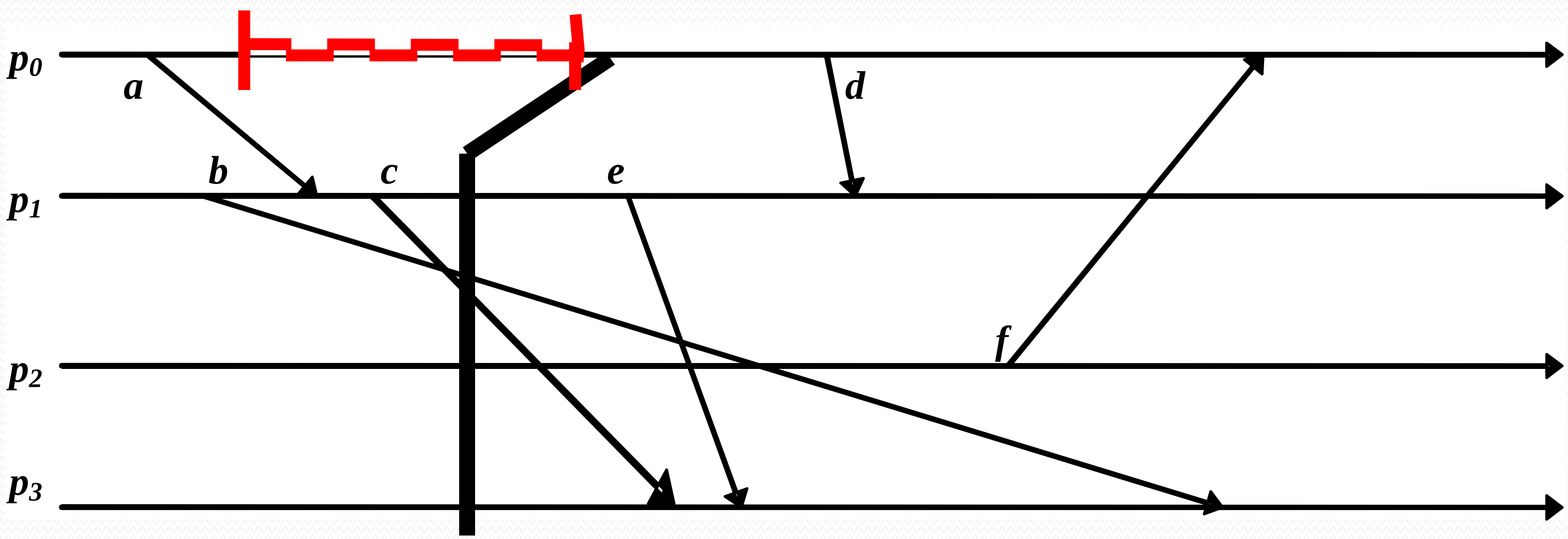
Consider…

- The picture we drew represents reality, but
  - With the same inputs, perhaps scheduling or contention on the machines could slow some down, or speed some up
  - Messages may be lost and need to be retransmitted, or might hit congested links
  - Or perhaps those problems occurred in the run in the picture but have gone away now

- In fact a given system might yield MANY pictures of this sort, depending on "luck"…

Temporal distortions

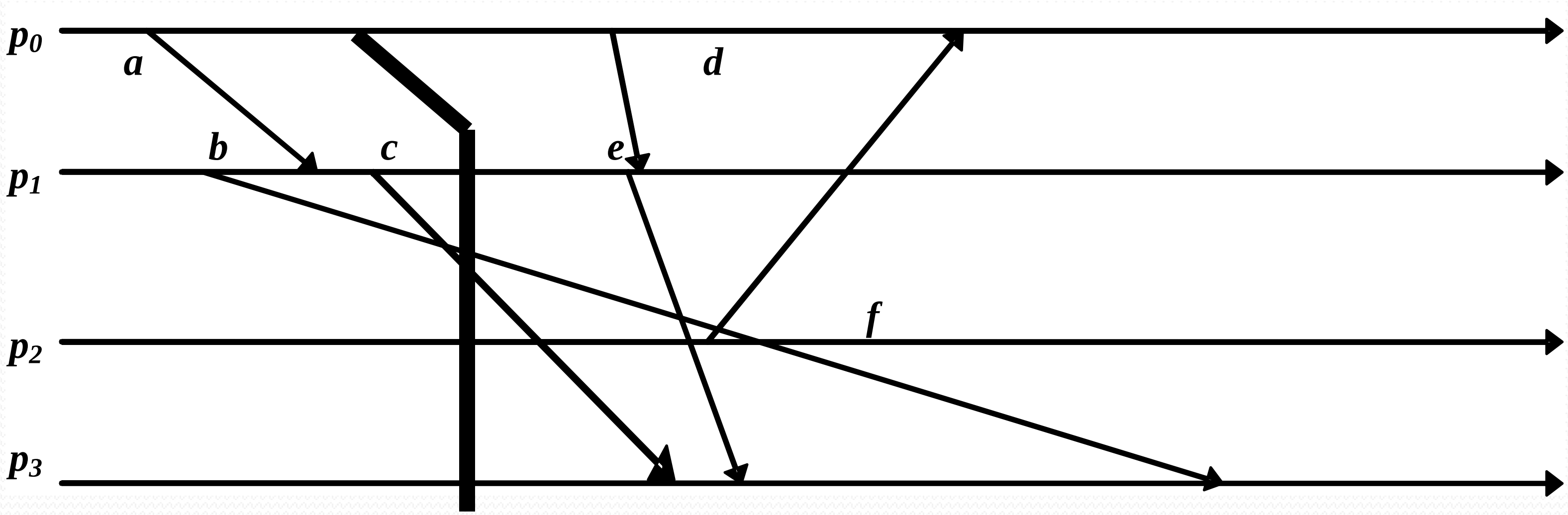
- Timelines can “stretch”…



- … caused by scheduling effects, message delays, message loss…

Temporal distortions

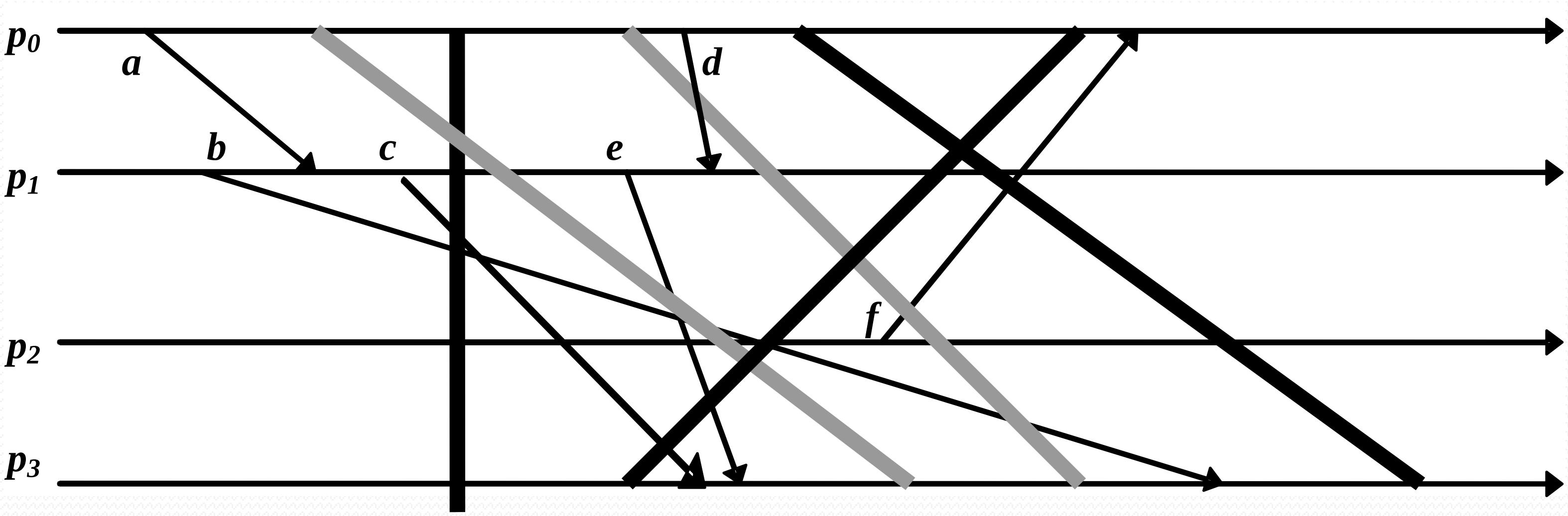
- Timelines can “shrink”



- E.g. something lets a machine speed up

Temporal distortions

- *Cuts* represent instants of time.



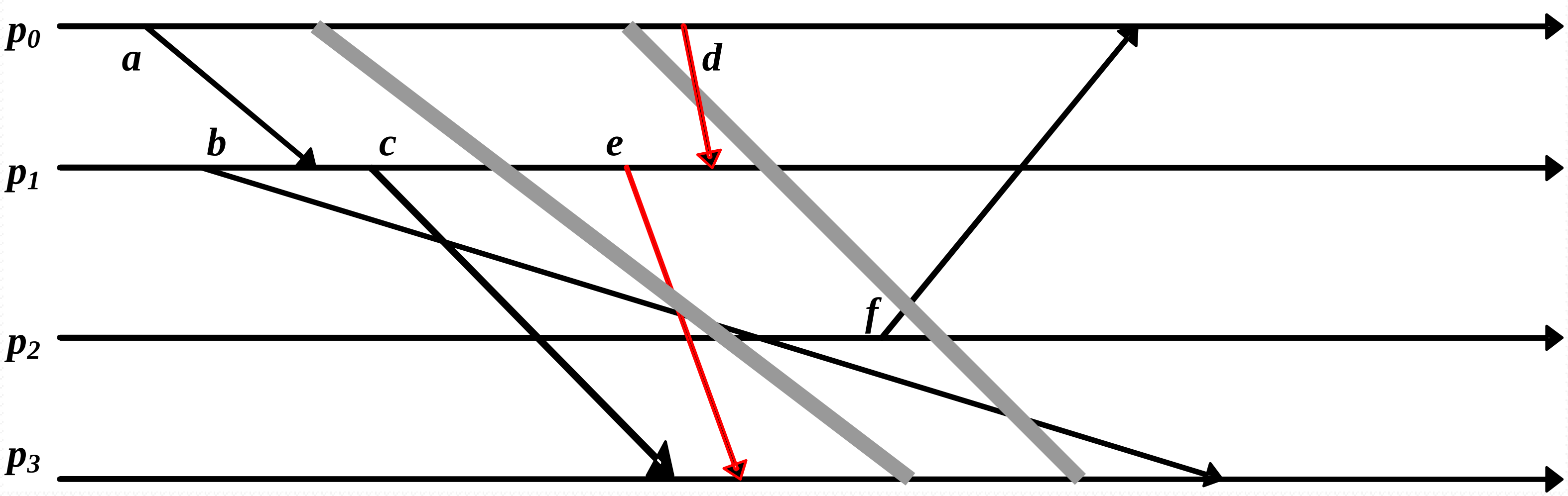
- But not every “cut” makes sense
  - Black cuts could occur but not gray ones.

Consistent cuts and snapshots

- Idea is to identify system states that "might" have occurred in real-life
  - Need to avoid capturing states in which a message is received but nobody is shown as having sent it
  - This the problem with the gray cuts

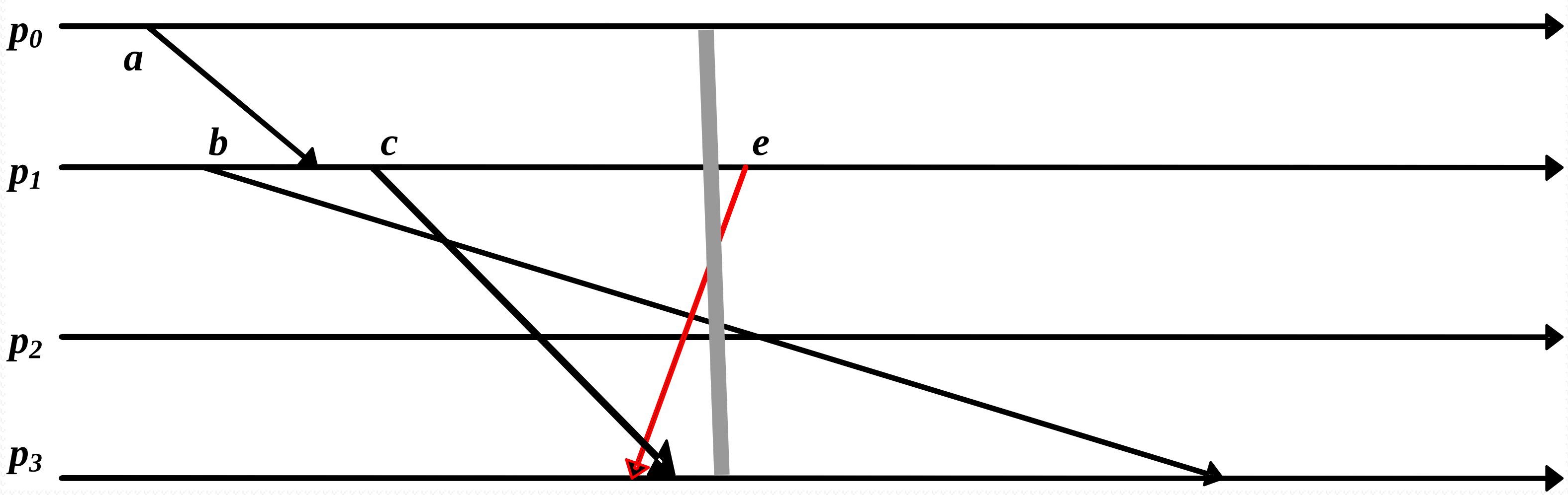
Temporal distortions

- Red messages cross gray cuts “backwards”



Temporal distortions

- Red messages cross gray cuts “backwards”



- In a nutshell: the cut includes a message that “was never sent”

Who cares?

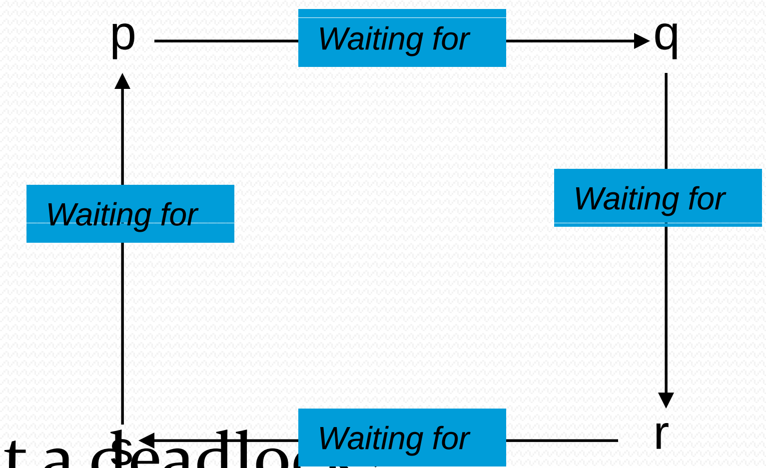
- In our auditing example, we might think some of the bank’s money is missing
- Or suppose that we want to do distributed deadlock detection
  - System lets processes “wait” for actions by other processes
  - A process can only do one thing at a time
  - A deadlock occurs if there is a circular wait

Deadlock detection "algorithm"

- p worries: perhaps we have a deadlock
- p is waiting for q, so sends "what's your state?"
- q, on receipt, is waiting for r, so sends the same question… and r for s…. And s is waiting on p.

Suppose we detect this state

- We see a cycle…

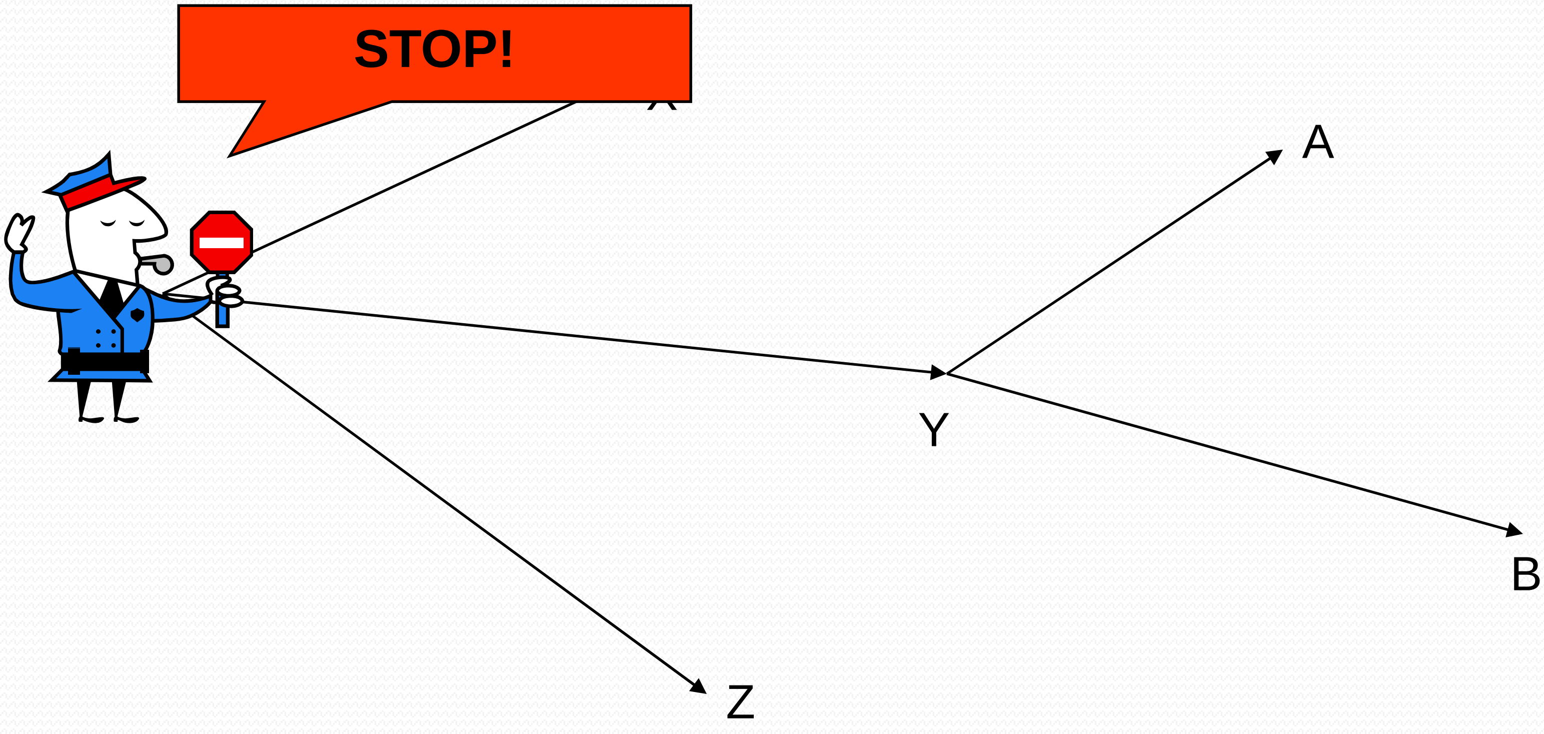


- … but is it a deadlock?

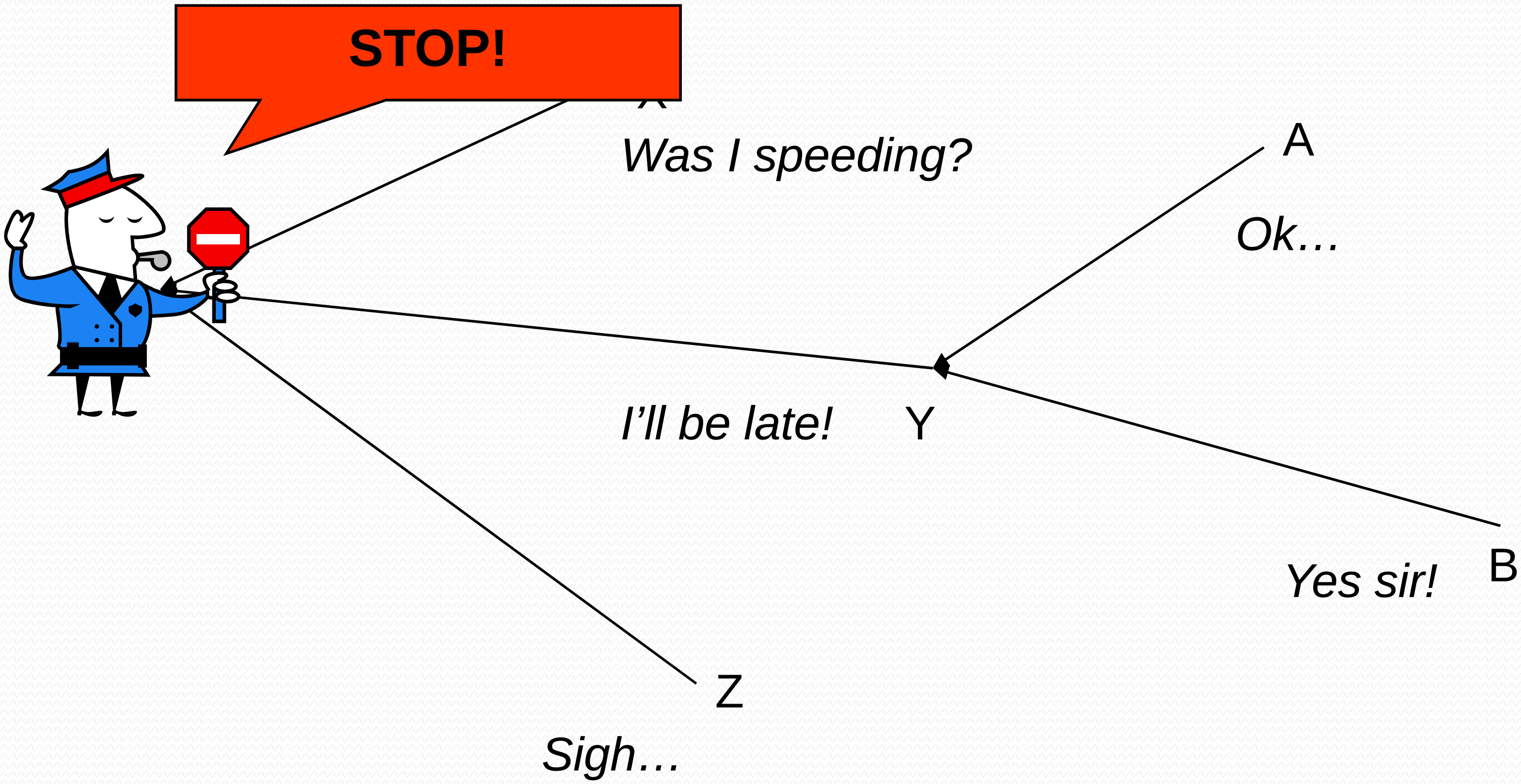
Phantom deadlocks!

- Suppose system has a *very high rate* of locking.
- Then perhaps a lock release message “passed” a query message
  - i.e. we see “q waiting for r” and “r waiting for s” but in fact, by the time we checked r, q was no longer waiting!
- In effect: we checked for deadlock on a gray cut – an inconsistent cut.

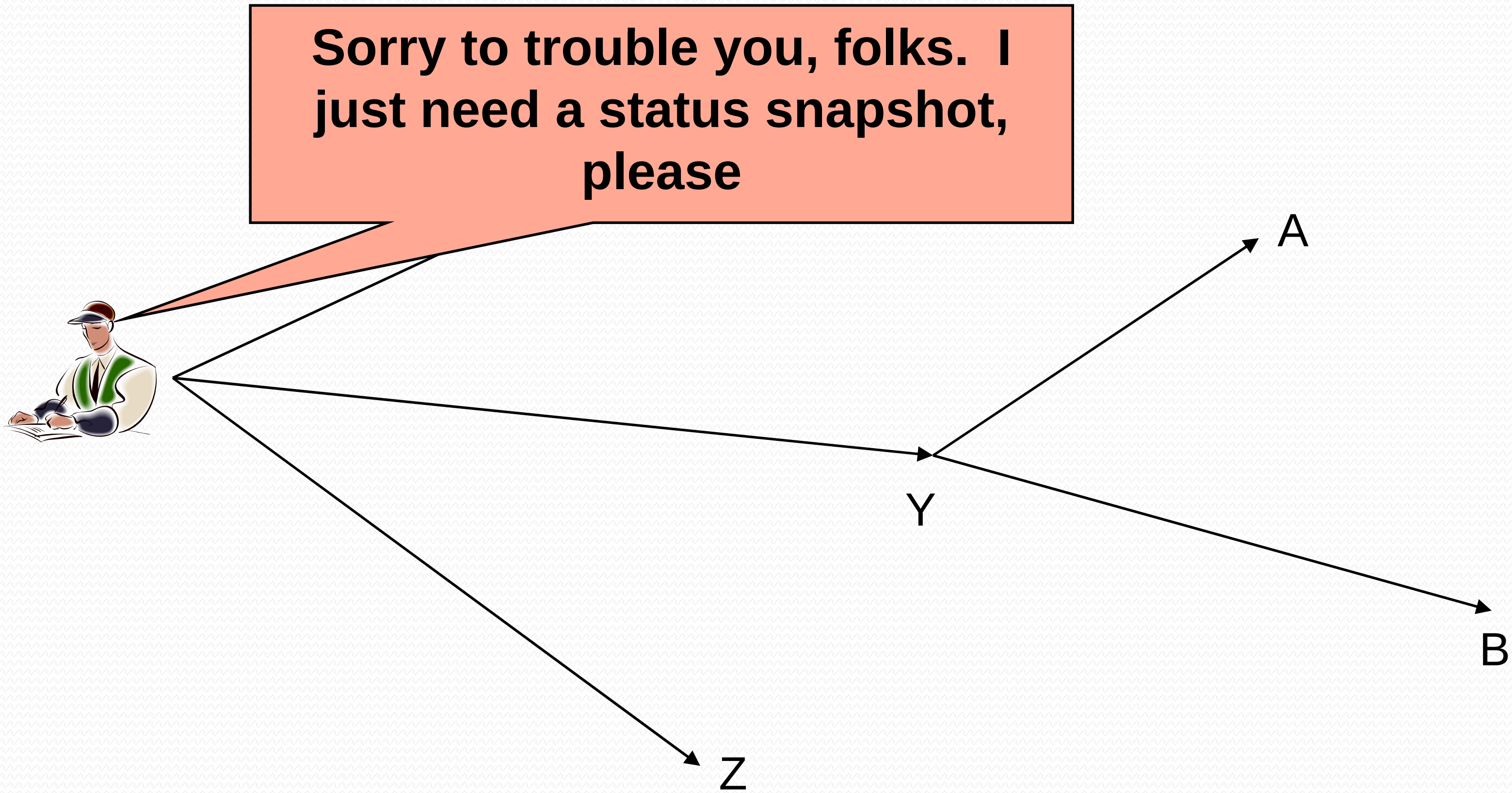
One solution is to "freeze" the system



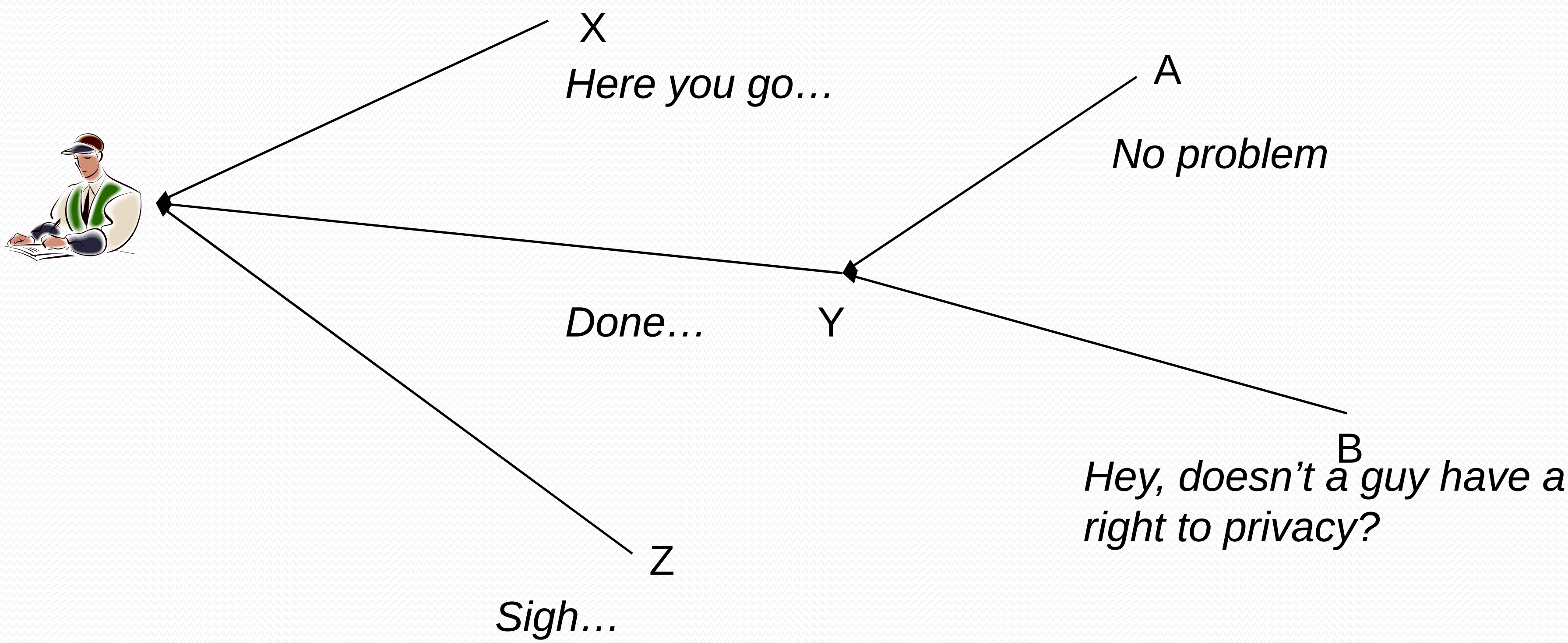
One solution is to “freeze” the system



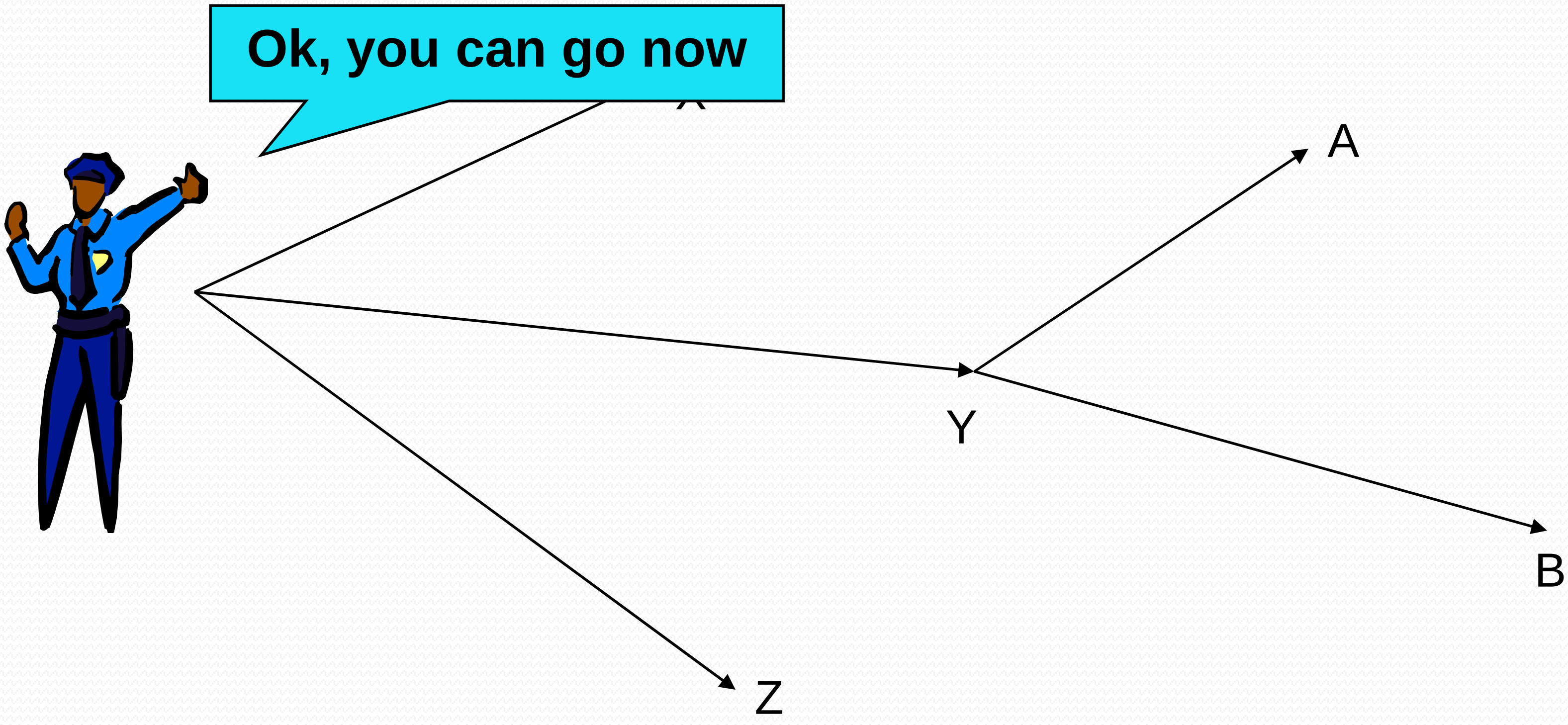
One solution is to "freeze" the system



One solution is to "freeze" the system



One solution is to "freeze" the system



Why does it work?

- When we check bank accounts, or check for deadlock, the system is idle
- So if “P is waiting for Q” and “Q is waiting for R” we really mean “simultaneously”
- But to get this guarantee we did something very costly because no new work is being done!

Consistent cuts and snapshots

- Goal is to draw a line across the system state such that
  - Every message "received" by a process is shown as having been sent by some other process
  - Some pending messages might still be in communication channels
- And we want to do this *while running*

Turn idea into an algorithm

- To start a new snapshot, p_i …
  - Builds a message: "P_i is initiating snapshot k".
    - The tuple (p_i, k) uniquely identifies the snapshot
  - Writes down its own state
  - Starts recording incoming messages on all channels

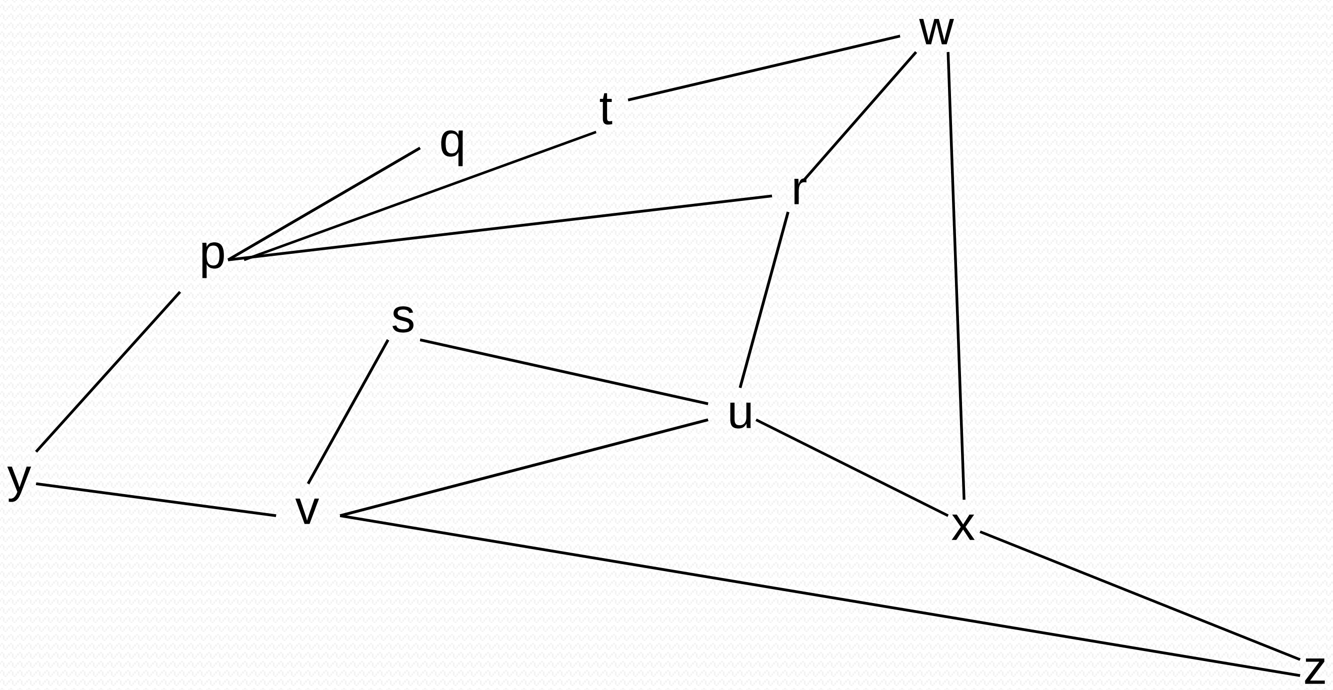
Turn idea into an algorithm

- Now p_i tells its neighbors to start a snapshot
- In general, on first learning about snapshot (p_i, k), p_x
  - Writes down its state: p_x's contribution to the snapshot
  - Starts "tape recorders" for all communication channels
  - Forwards the message on all outgoing channels
  - Stops "tape recorder" for a channel when a snapshot message for (p_i, k) is received on it
- Snapshot consists of all the local state contributions and all the tape-recordings for the channels

Chandy/Lamport

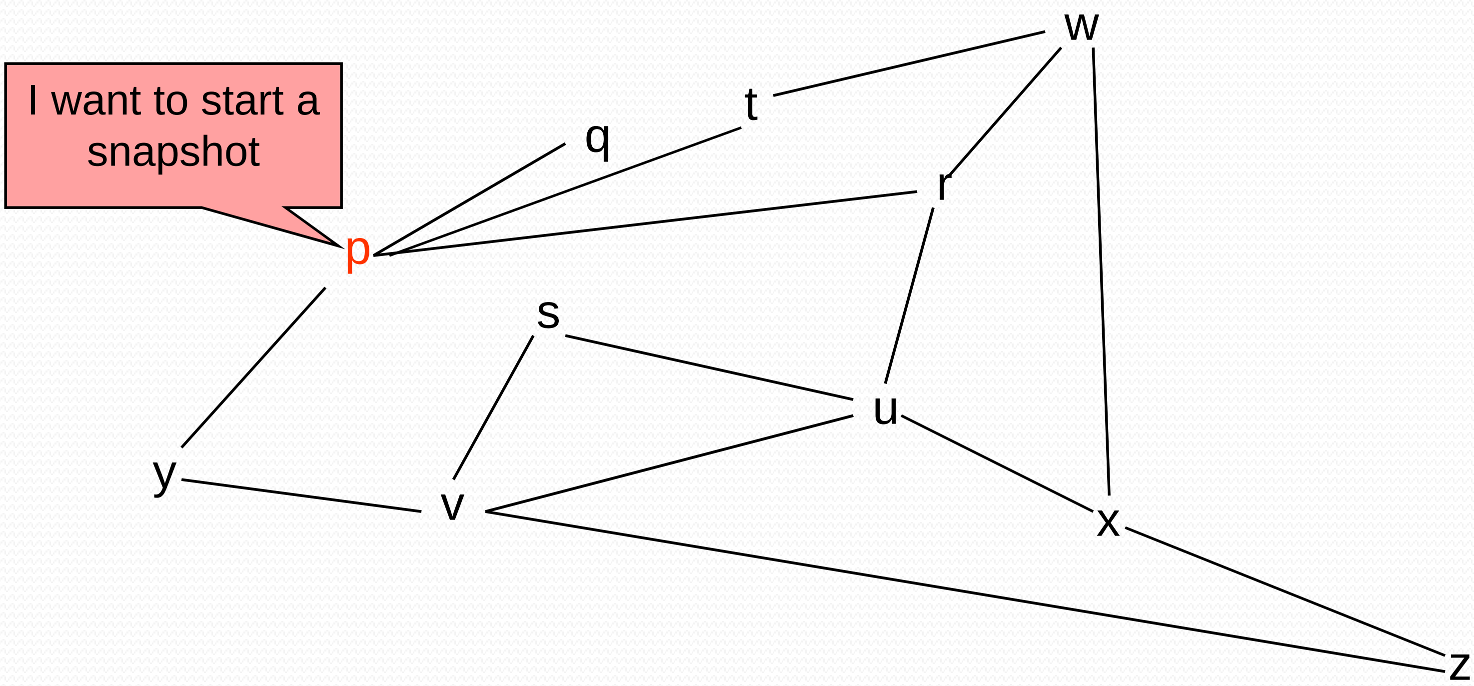
- Outgoing wave of requests… incoming wave of snapshots and channel state
- Snapshot ends up accumulating at the initiator, p_i
- Algorithm doesn’t tolerate process failures or message failures.

Chandy/Lamport



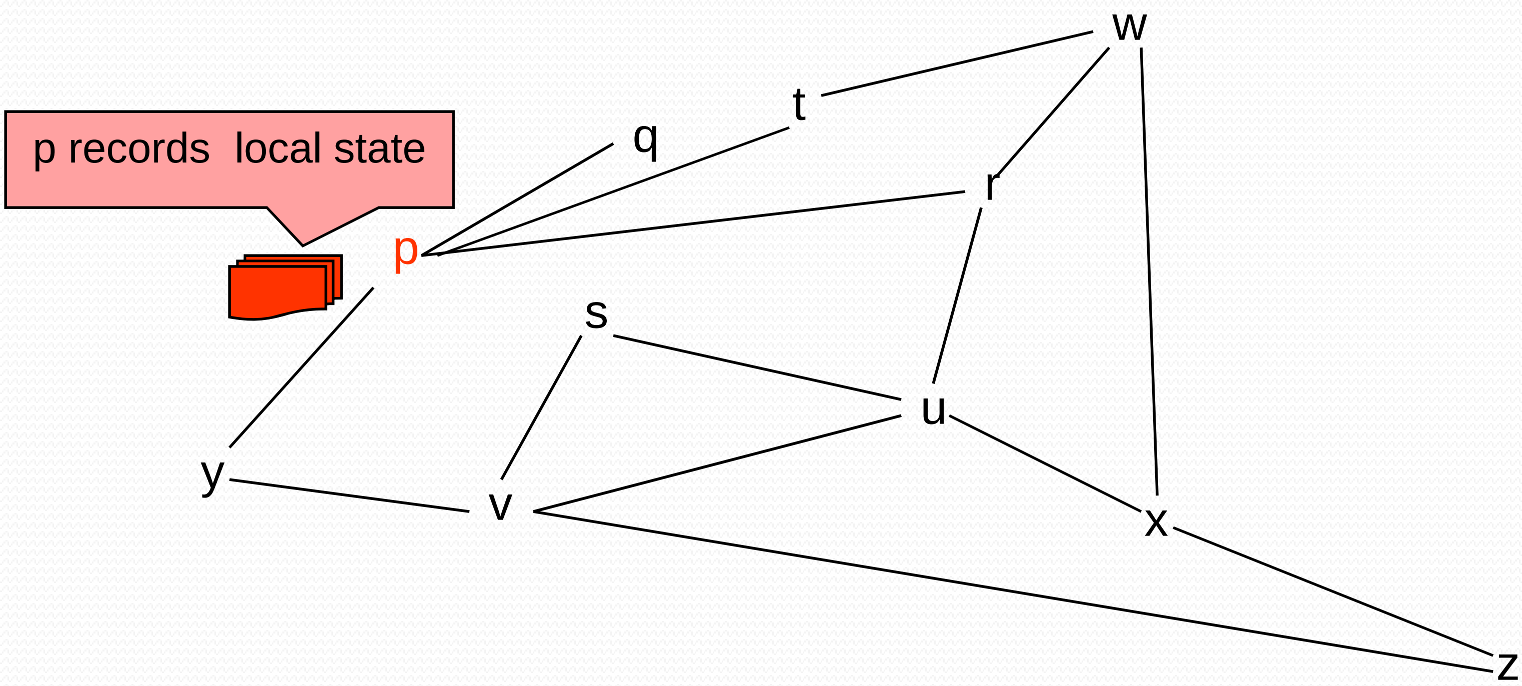
A network

Chandy/Lamport



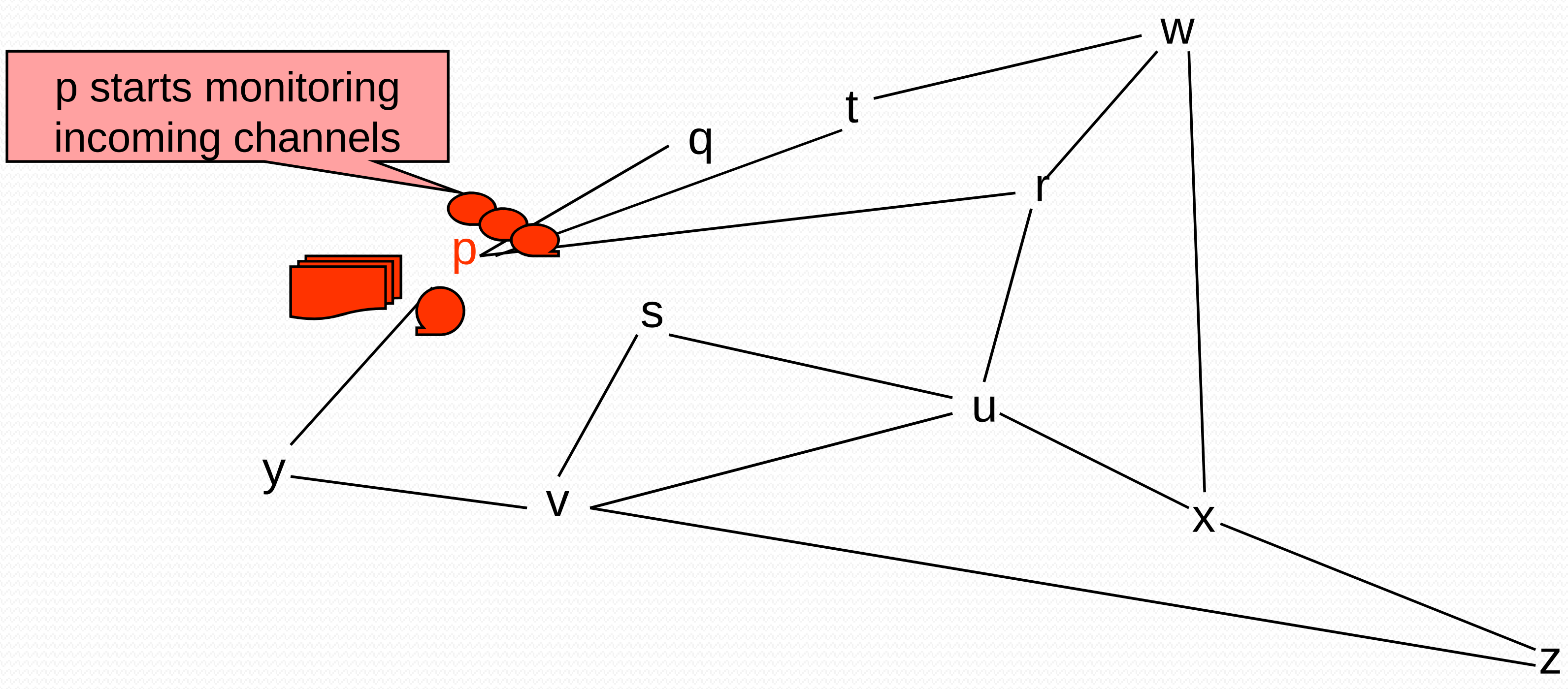
A network

Chandy/Lamport



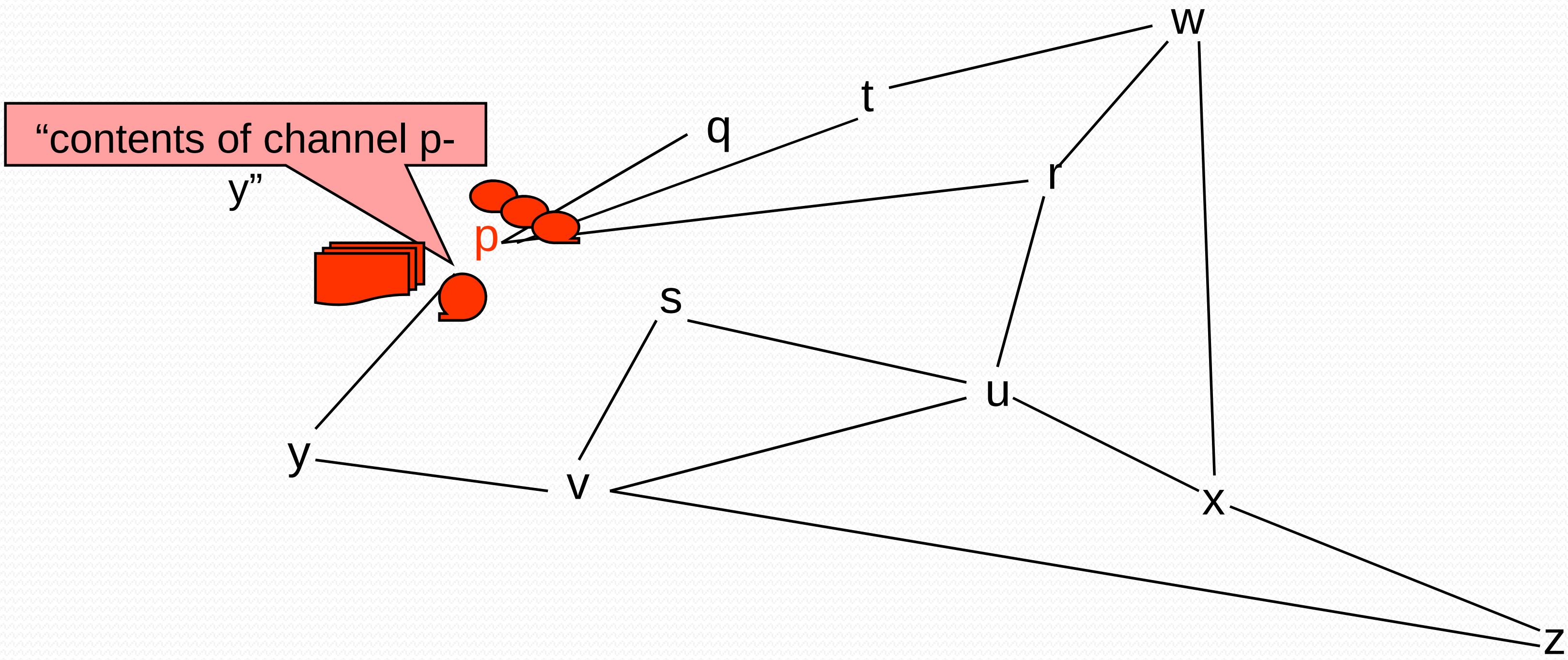
A network

Chandy/Lamport



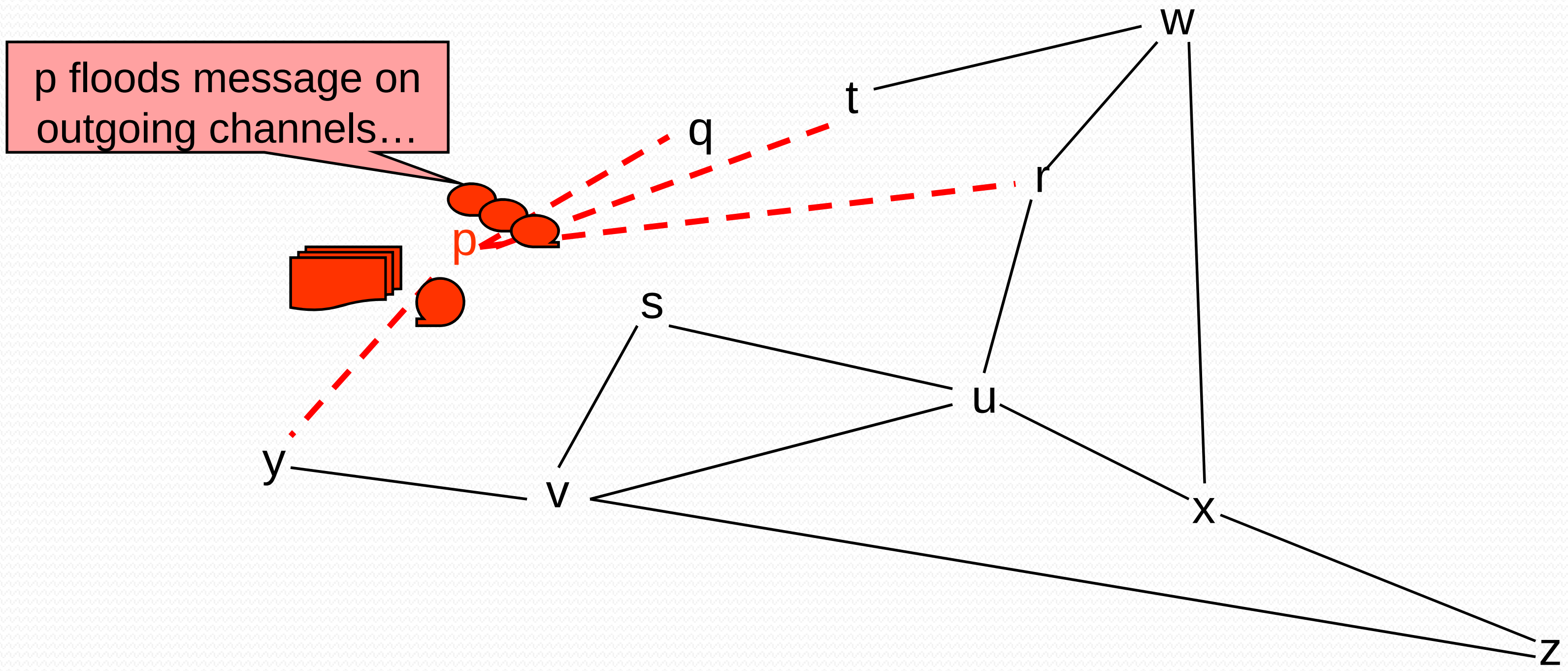
A network

Chandy/Lamport



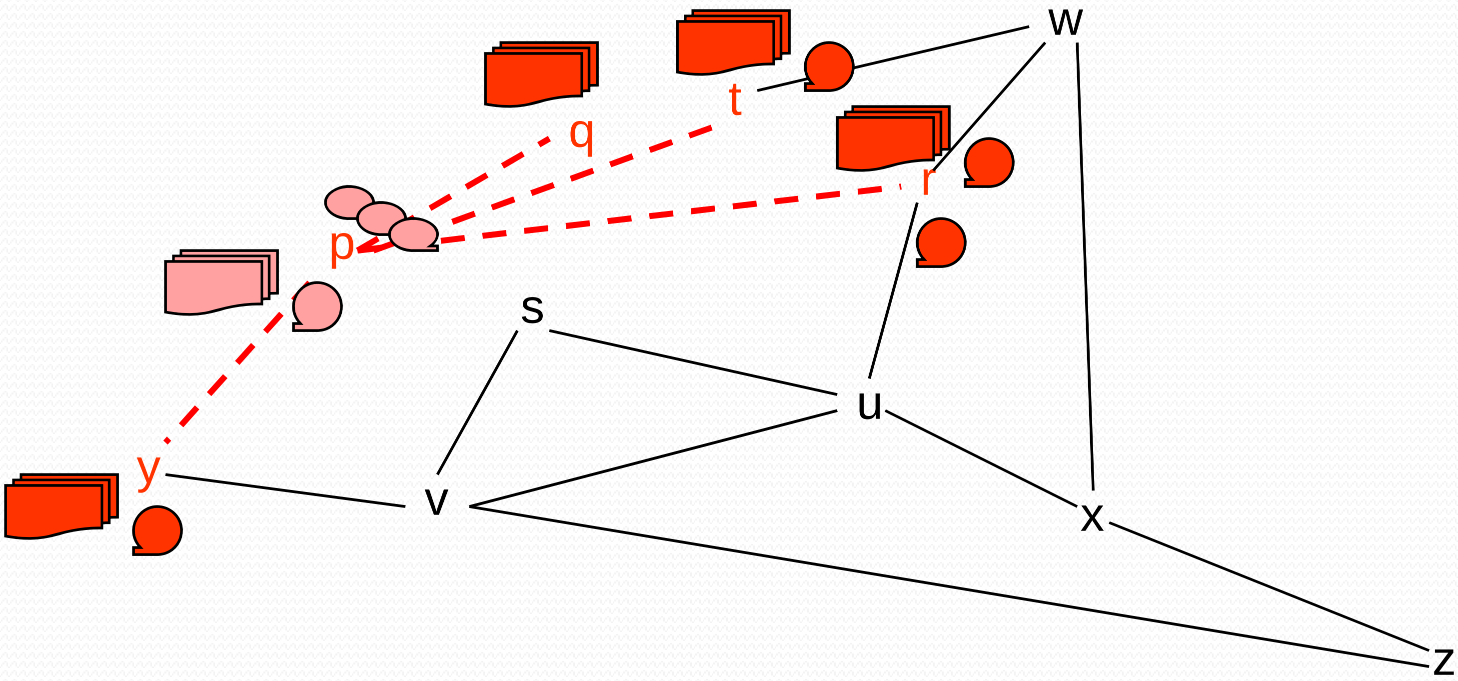
A network

Chandy/Lamport



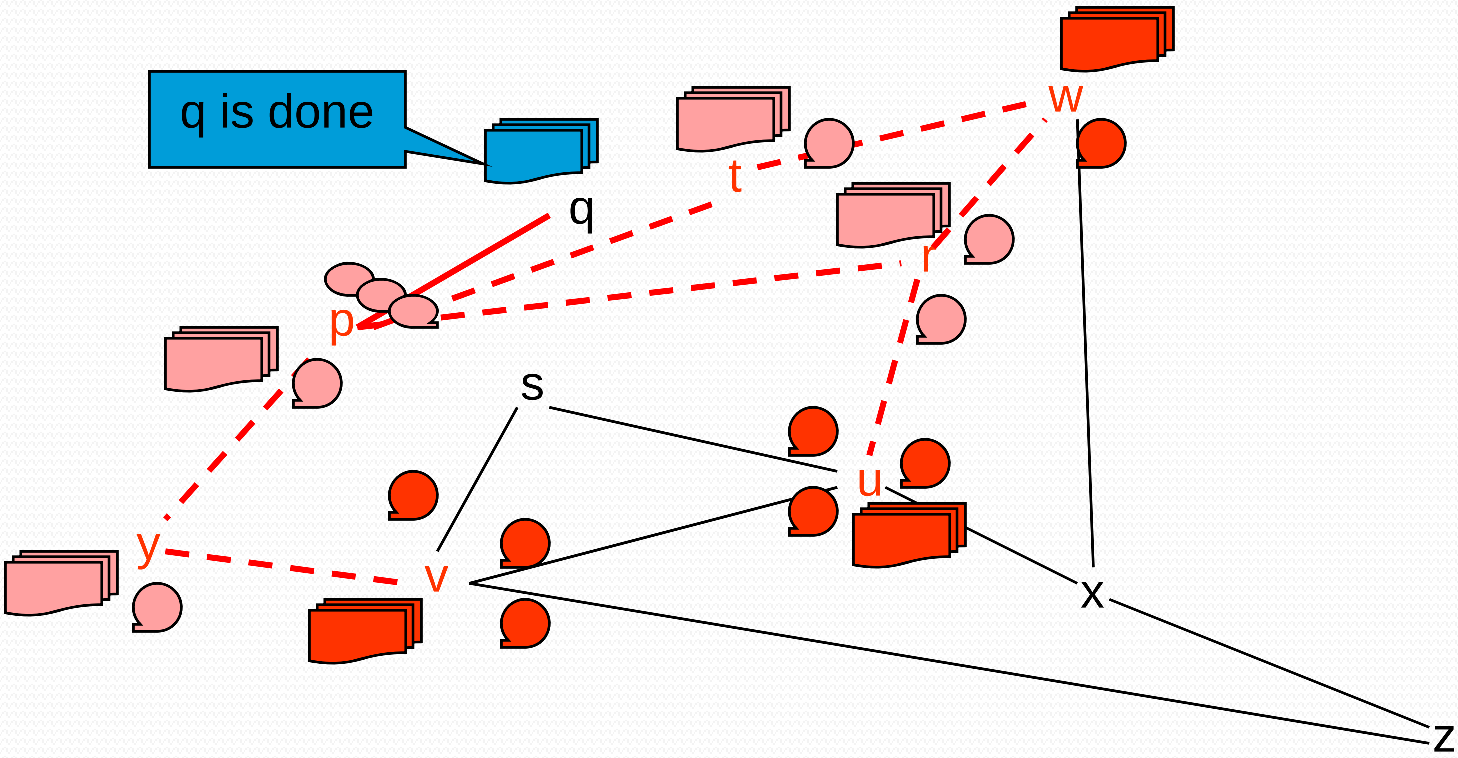
A network

Chandy/Lamport



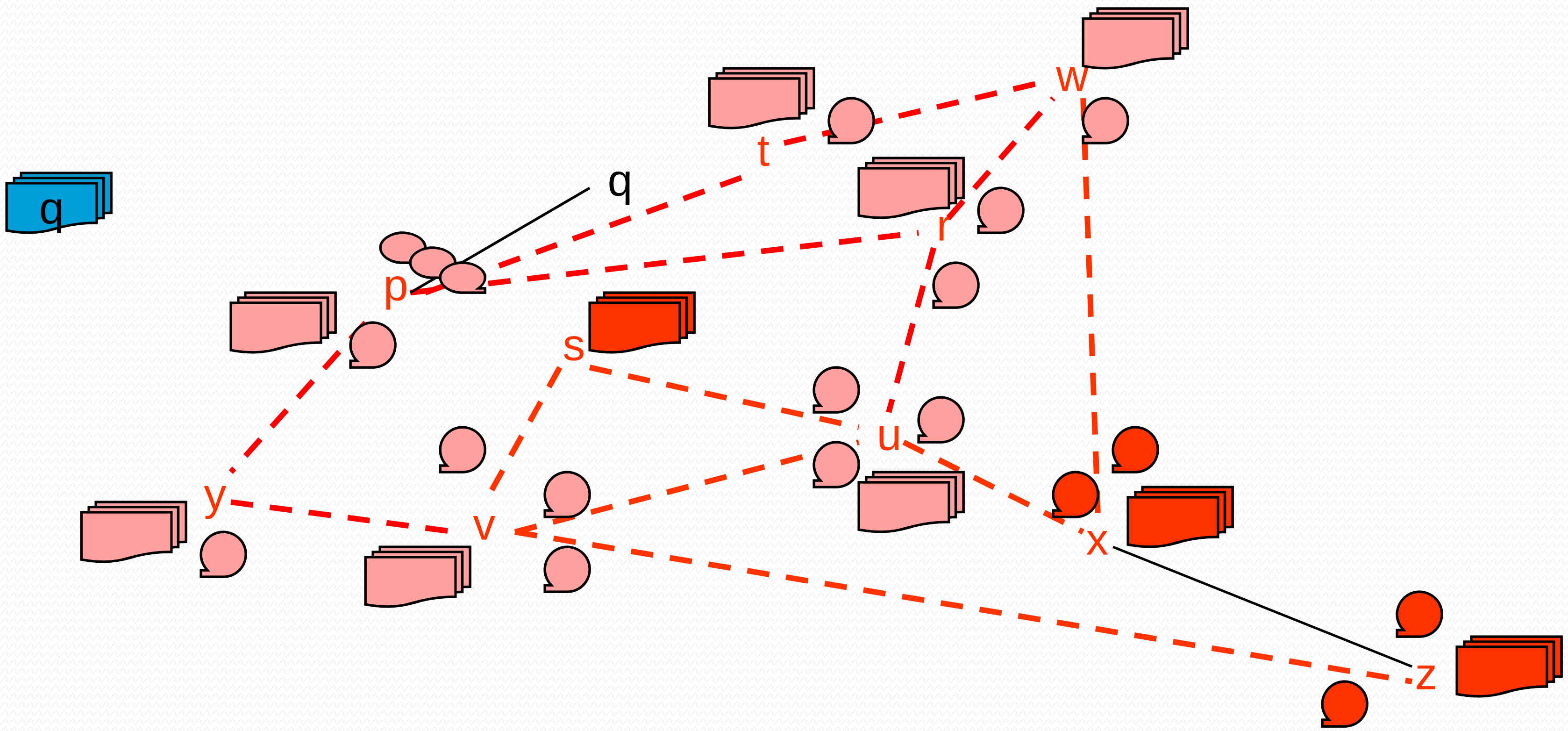
A network

Chandy/Lamport



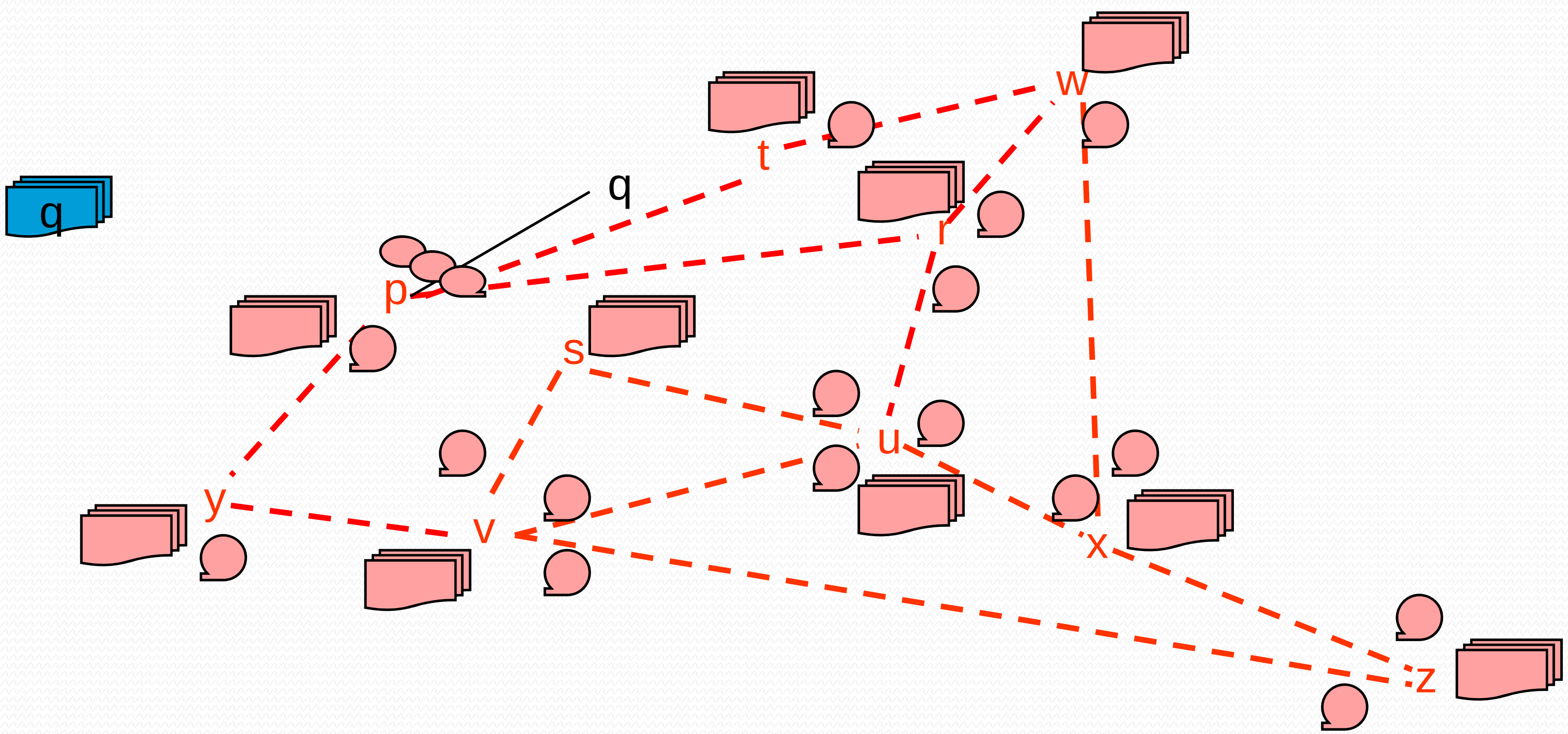
A network

Chandy/Lamport



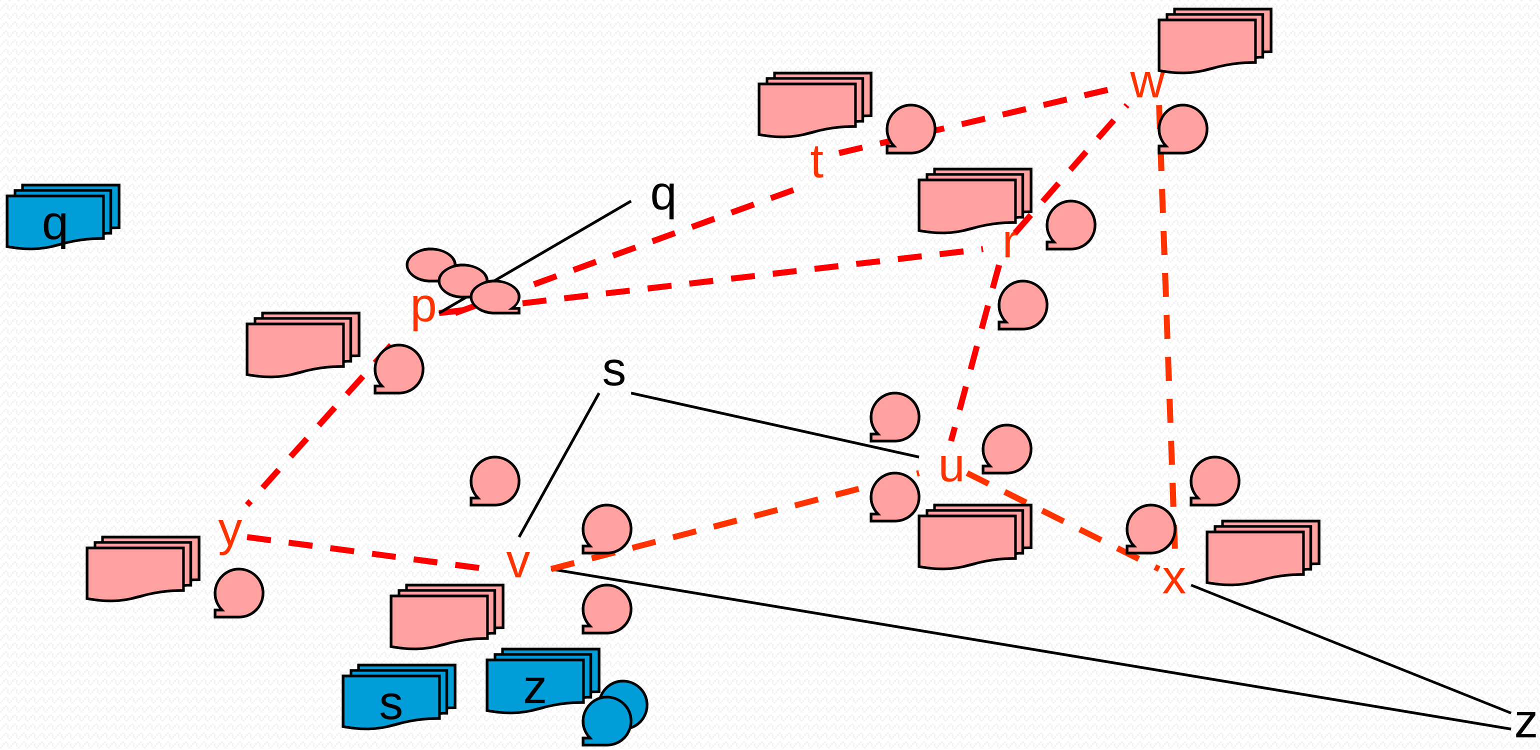
A network

Chandy/Lamport



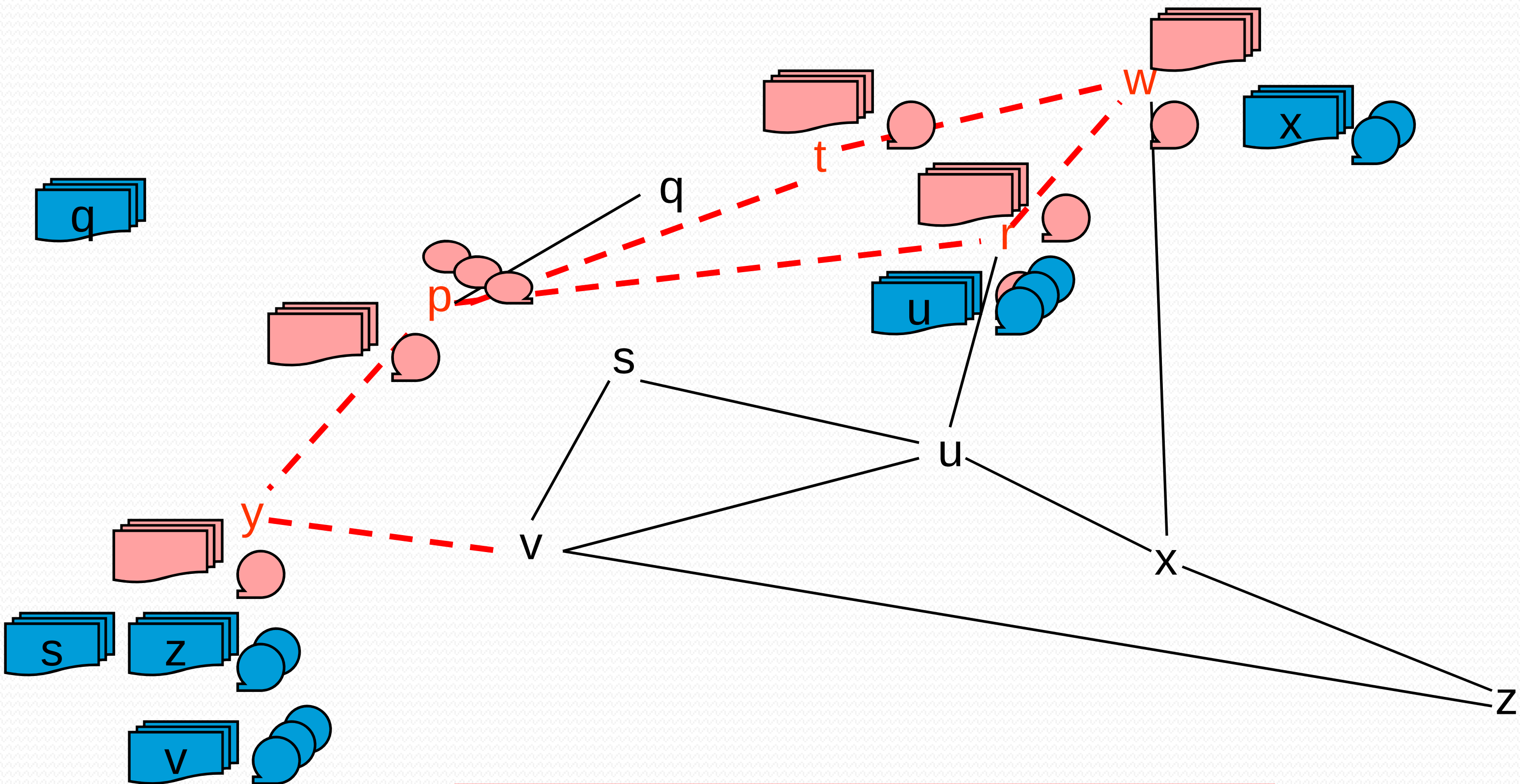
A network

Chandy/Lamport



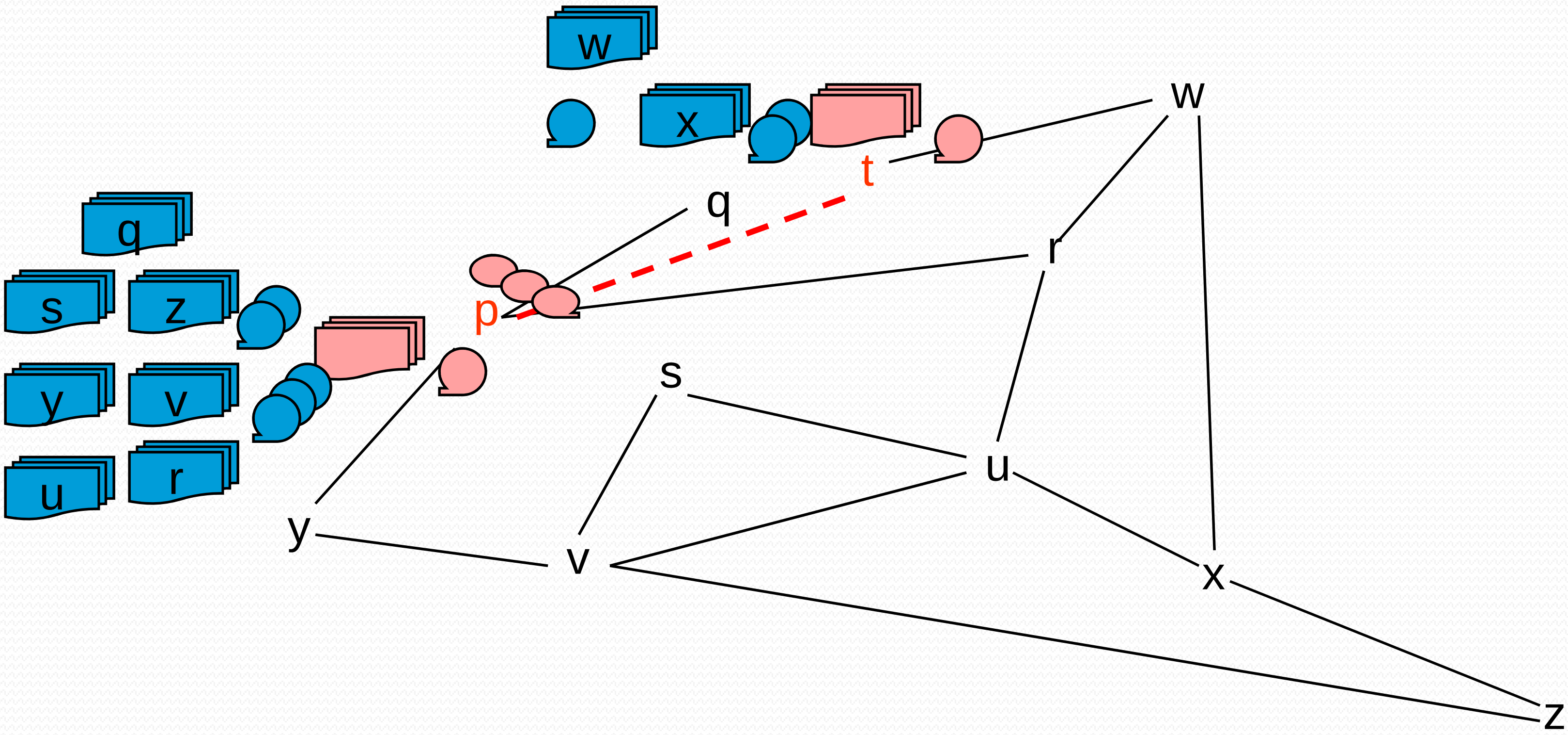
A network

Chandy/Lamport



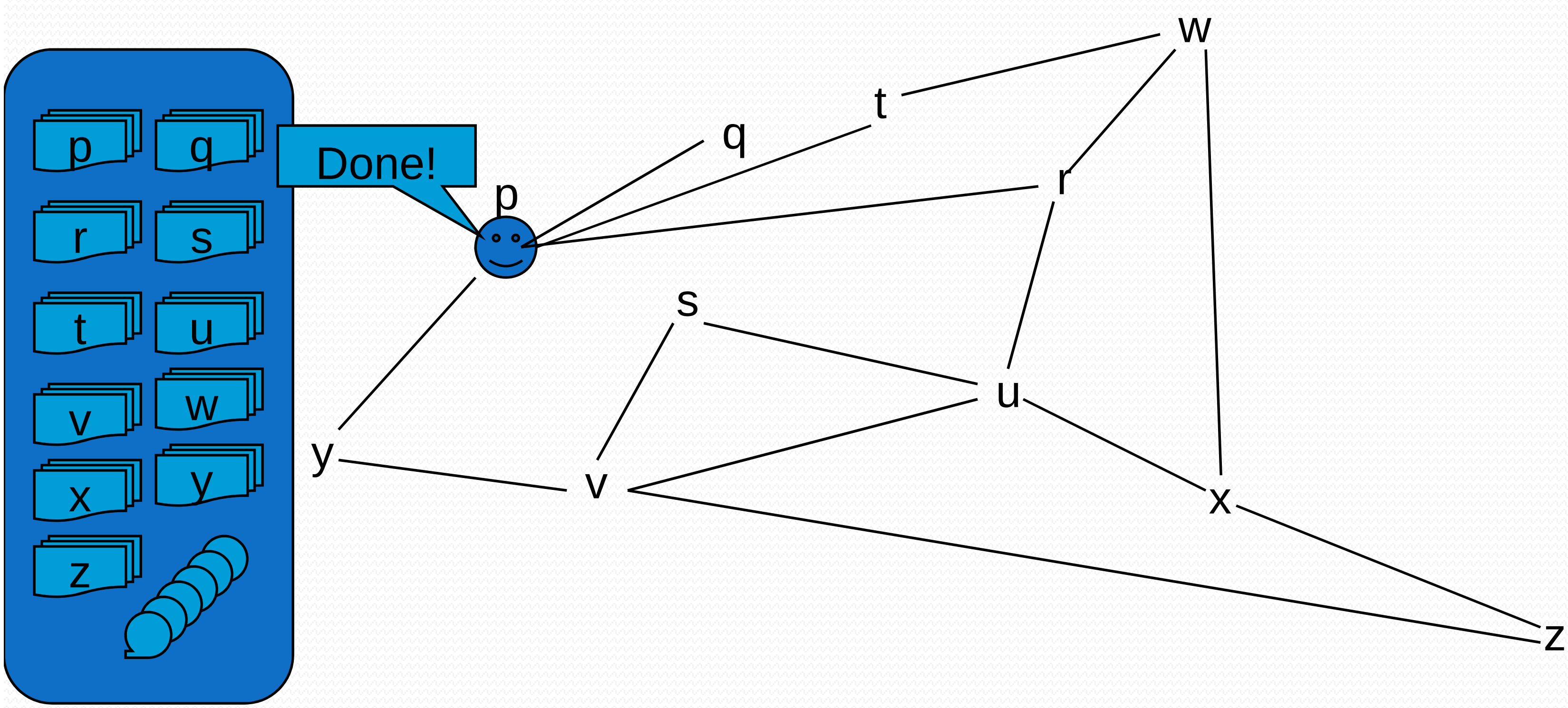
A network

Chandy/Lamport



A network

Chandy/Lamport



A snapshot of a network

Using logical clocks for cuts

- Application could also set a logical clock WAY ahead
- Rule: *each time the clock reaches a multiple of 100,000,000 write down your state*
  - So: node p sets clock ahead to 1,000,001 (and writes down its state). Then floods the network
  - As the message reaches nodes, each records its state

Summary

- We've seen that true clocks are "tricky" in distributed systems but that we can use simple integers or vectors of integers to capture event ordering
  - Logical clocks capture just part of the ordering
  - Vector clocks are larger but capture all the useful info.
- Then we looked at how one can interpret "simultaneous" as a distributed concept
  - Consistent snapshots or cuts (cuts being the "front line" of a snapshot, which includes channel state too)