CS5412: REPLICATION, CONSISTENCY AND CLOCKS

Lecture X

Ken Birman

Recall that clouds have tiers

- Up to now our focus has been on client systems and the network, and the way that the cloud has reshaped both
- We looked very superficially at the tiered structure of the cloud itself
  - Tier 1: Very lightweight, responsive "web page builders" that can also route (or handle) "web services" method invocations. Limited to "soft state".
  - Tier 2: (key,value) stores and similar services that support tier 1. Basically, various forms of caches.
  - Inner tiers: Online services that handle requests not handled in the first tier. These can store persistent files, run transactional services. But we shield them from load.
  - Back end: Runs offline services that do things like indexing the web overnight for use by tomorrow morning's tier-1 services.

Replication

- A central feature of the cloud
- To handle more work, make more copies
  - In the first tier, which is highly elastic, data center management layer pre-positions inactive copies of virtual machines for the services we might run
    - Exactly like installing a program on some machine
  - If load surges, creating more instances just entails
    - Running more copies on more nodes
    - Adjusting the load-balancer to spray requests to new nodes
  - If load drops… just kill the unwanted copies!
    - Little or no warning. Discard any "state" they created locally.

Replication is about keeping copies

- The term may sound fancier but the meaning isn't

- Whenever we have many copies of something we say that we've replicated that thing
  - But usually replica does connote "identical"
  - Instead of *replication* we use the term *redundancy* for things like alternative communication paths (e.g. if we have two distinct TCP connections from some client system to the cloud)
  - Redundant things might not be identical. Replicated things usually play identical roles and have equivalent data.

Things we can replicate in a cloud

- Files or other forms of data used to handle requests
  - If all our first tier systems replicate the data needed for end-user requests, then they can handle all the work!
  - Two cases to consider: in one the data itself is "write once" like a photo. Either you have a replica, or don't
  - In the other the data evolves over time, like the current inventory count for the latest iPad in the Apple store
- Computation
  - Here we replicate some *request* and then the work of computing the answer can be spread over multiple programs in the cloud
  - We benefit from parallelism by getting a faster answer
  - Can also provide fault-tolerance

Many things "map" to replication

- As we just saw, data (or databases), computation
- Fault-tolerant request processing
- Coordination and synchronization (e.g. "who's in charge of the air traffic control sector over Paris?")
- Parameters and configuration data
- Security keys and lists of possible users and the rules for who is permitted to do what
- Membership information in a DHT or some other service that has many participants

So... focus on replication!

- If we can get replication right, we'll be on the road to a highly assured cloud infrastructure
- Key is to understand what it means to correctly replicate data at cloud scale...
- ... then once we know what we *want* to do, to find scalable ways to implement needed abstraction(s)

Concept of “consistency”

- We would say that a replicated entity behaves in a consistent manner if mimics the behavior of a non-replicated entity
  - E.g. if I ask it some question, and it answers, and then you ask it that question, your answer is either the same or reflects some update to the underlying state
  - Many copies but acts like just one

- An inconsistent service is one that seems “broken”

Consistency lets us ignore implementation

A <u>consistent</u> distributed system will often have many components, but users observe behavior indistinguishable from that of a single-component reference system

Reference Model

Implementation

Dangers of Inconsistency

My rent check bounced? That can't be right!

- Inconsistency causes bug
  - Clients would never be able to trust servers… a free-for-all

- Weak or "best effort" consistency?
  - Common in today's cloud replication schemes
  - But strong security guarantees demand consistency
  - Would you trust a medical electronic-health records system or a bank that used "weak consistency" for better scalability?

Leslie Lamport's insight

- To formalize notions of consistency, start by formalizing notions of time
- Once we do this we can be rigorous about notions like "before" or "after" or "simultaneously"
  - If we try to write down conditions for correct replication these kinds of terms often arise

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

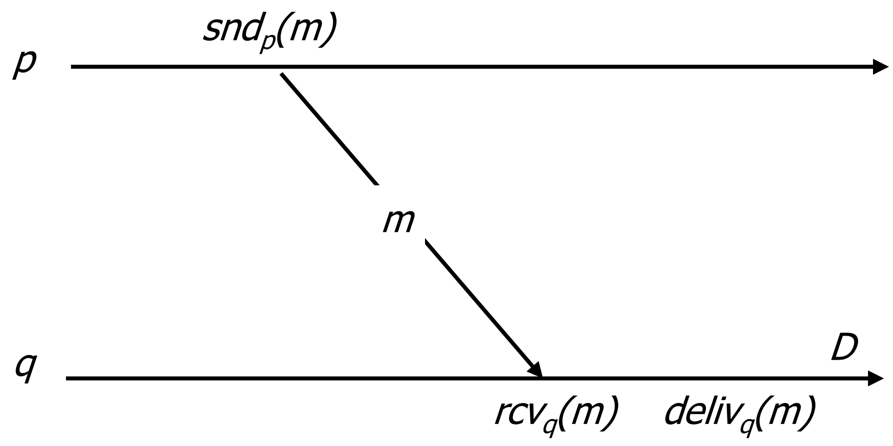
But what does time “mean”?

- Time on a global clock?
  - E.g. on Cornell clock tower?
  - … or perhaps on a 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

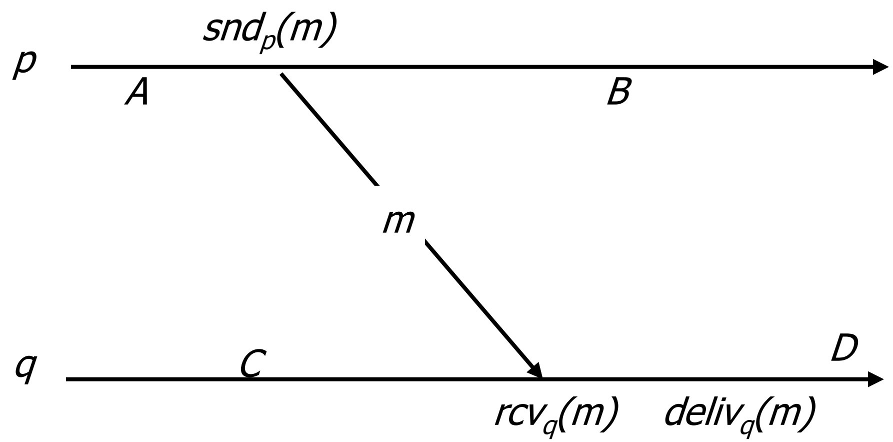
Lamport's approach

- 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:

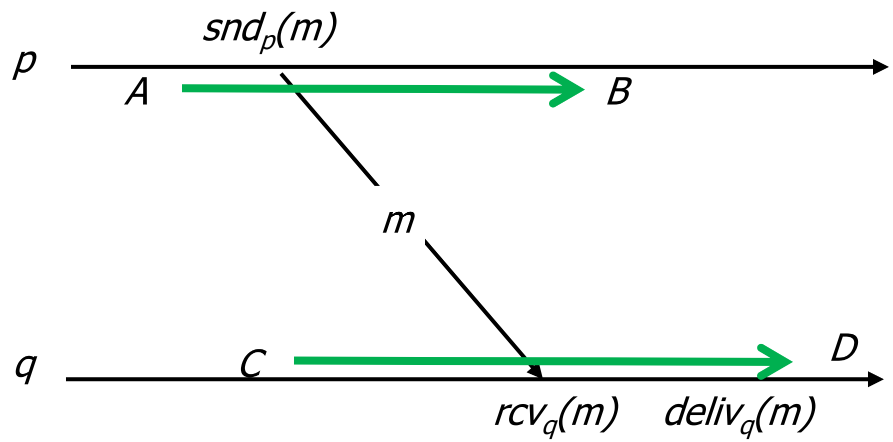


Drawing time-line pictures:



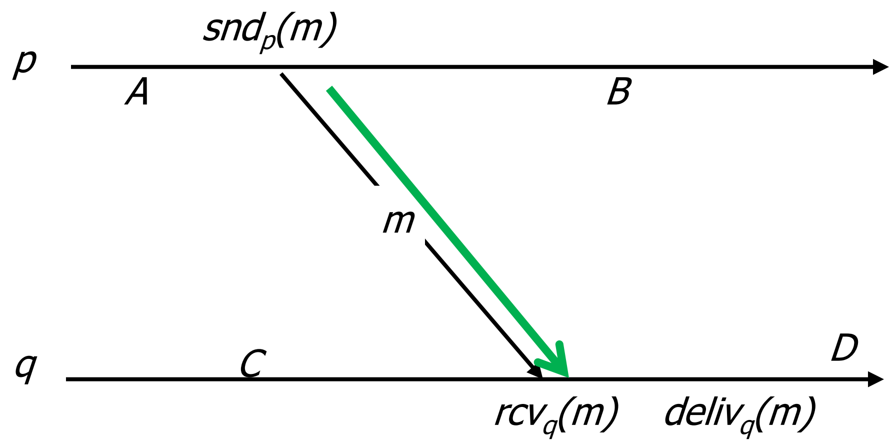
- A, B, C and D are "events".
  - Could be anything meaningful to the application
  - So are snd(m) and rcv(m) and deliv(m)
- What ordering claims are meaningful?

Drawing time-line pictures:



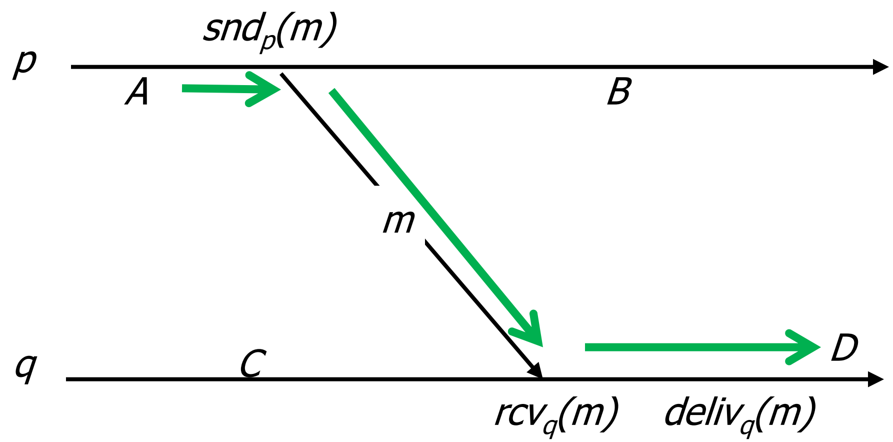
- A happens before B, and C before D
  - "Local ordering" at a single process
  - Write $A \xrightarrow{p} B$ and $C \xrightarrow{q} D$

Drawing time-line pictures:



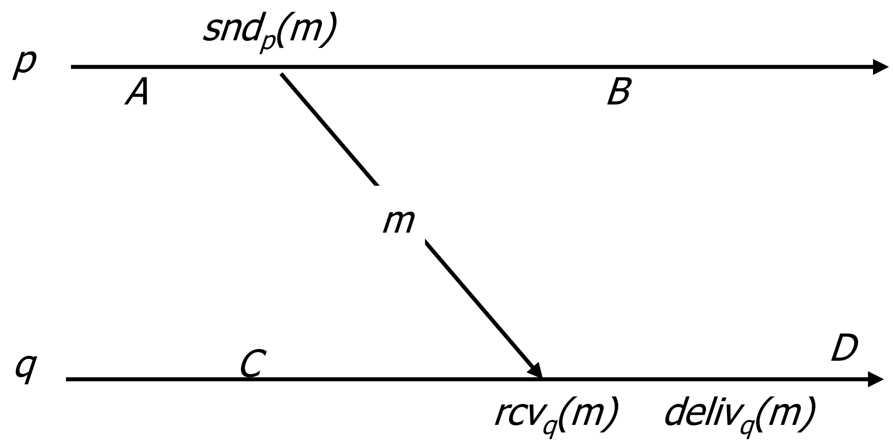
- $snd_p(m)$ also happens before $rcv_q(m)$
  - "Distributed ordering" introduced by a message
  - Write $snd_p(m) \xrightarrow{M} rcv_q(m)$

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 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 transitive closure of rules (1) and (2)

- Notice that, so far, this is just a mathematical notation, not a "systems tool"
  - Given a trace of what happened in a system we could use these tools to talk about the trace
  - But need a way to "implement" this idea

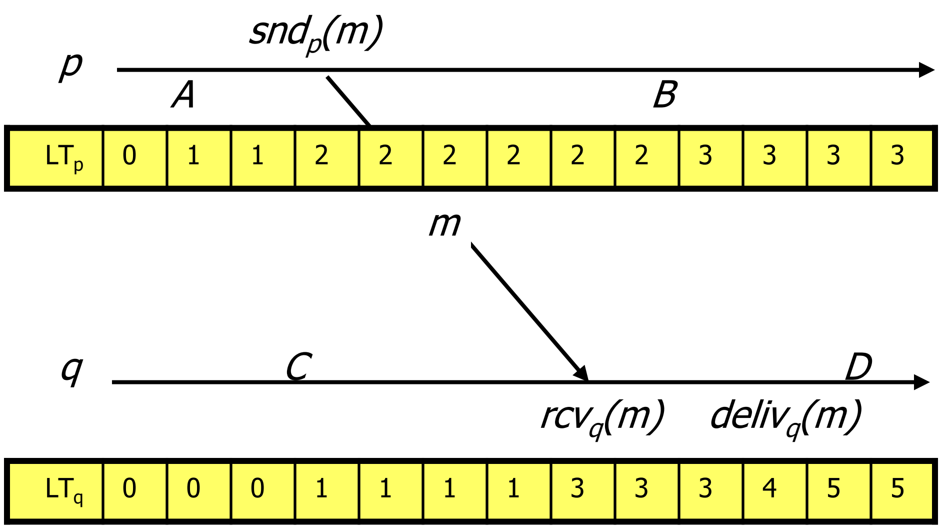
Logical clocks

- 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_p	0	1	1	2	2	2	2	2	2	3	3	3	3
LT_q	0	0	0	1	1	1	1	3	3	3	4	5	5

- 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→B, then LT(A)<LT(B)
- But converse might not be true:
  - If LT(A)<LT(B) can't be sure that A→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

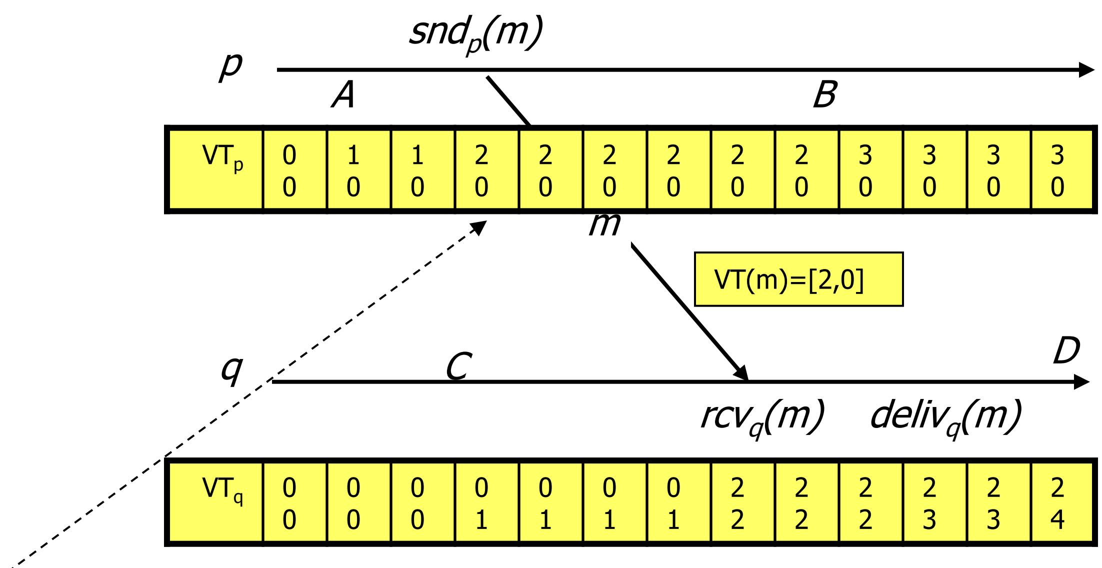
History of vector clocks?

- Originated in work at UCLA on file systems that allowed updates from multiple sources concurrently
  - Jerry Popek's FICUS system
  - Today version systems (e.g. SVN, CVS) use the idea
- Also gradually adopted in distributed systems
- Most of the "formal" work was done by Fidge and Mattern in Europe, long after idea was in wide use

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

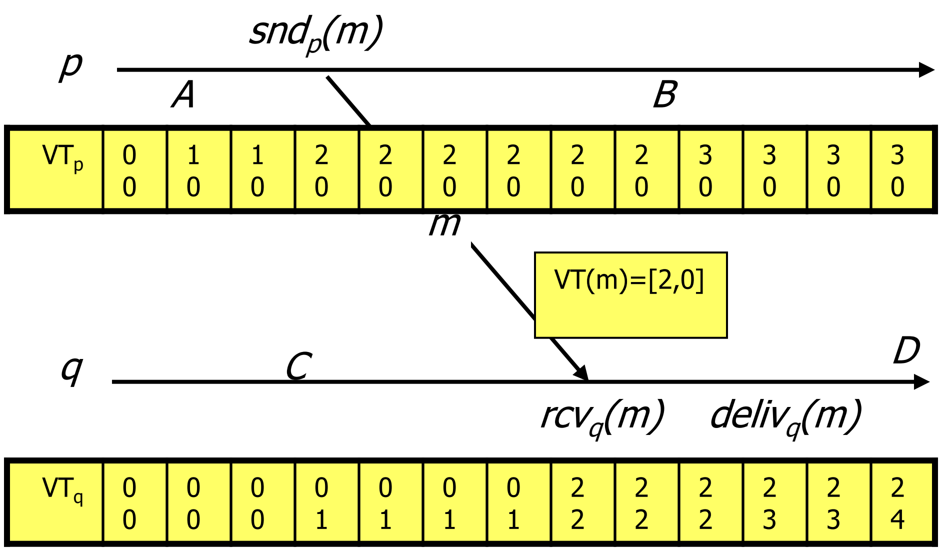


Could also be [1,0] if we decide not to increment the clock on a snd event. Decision depends on how the timestamps will be used.

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

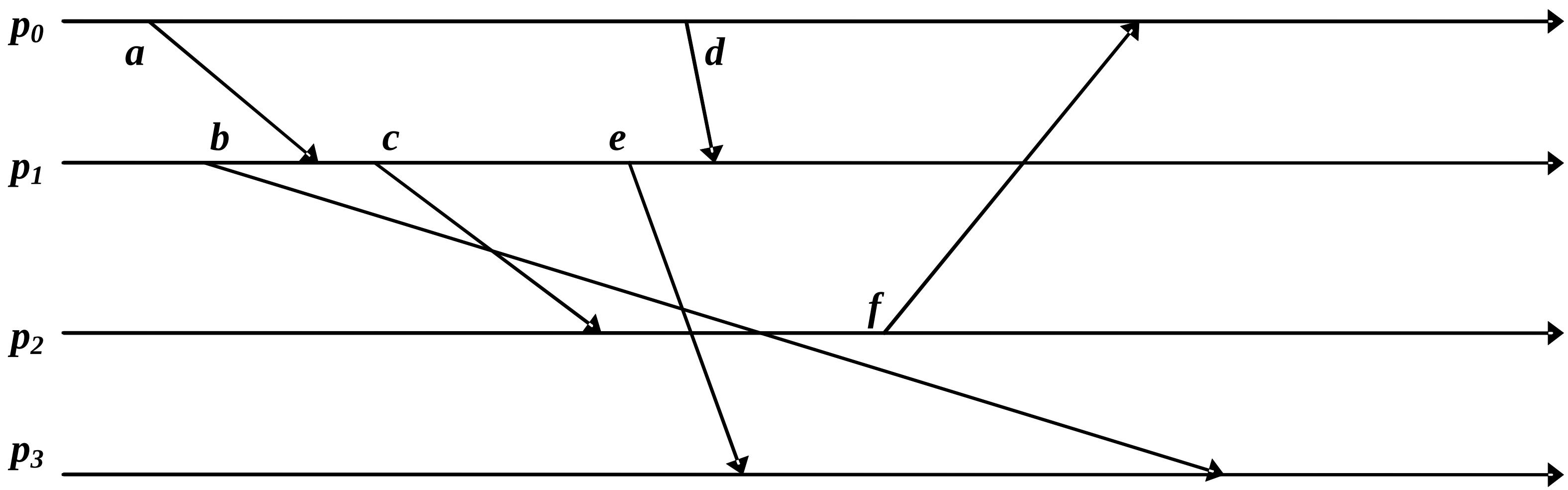
- If A→B, then VT(A)<VT(B)
  - Write a chain of events from A to B
  - Step by step the vector clocks get larger
- If VT(A)<VT(B) then A→B
  - Two cases: if A and B both happen at same process p, trivial
  - If A happens at p and B at q, can trace the path back by which q “learned” $VT_A[p]$
- Otherwise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

Temporal 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

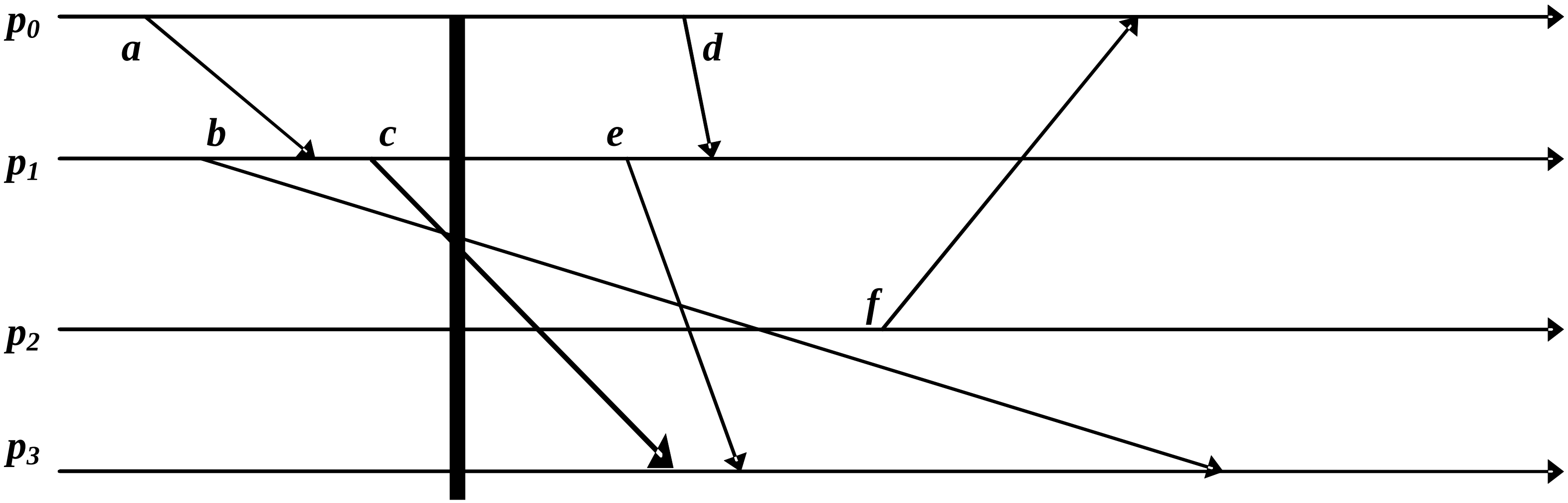
Temporal distortions

- What does "now" mean?



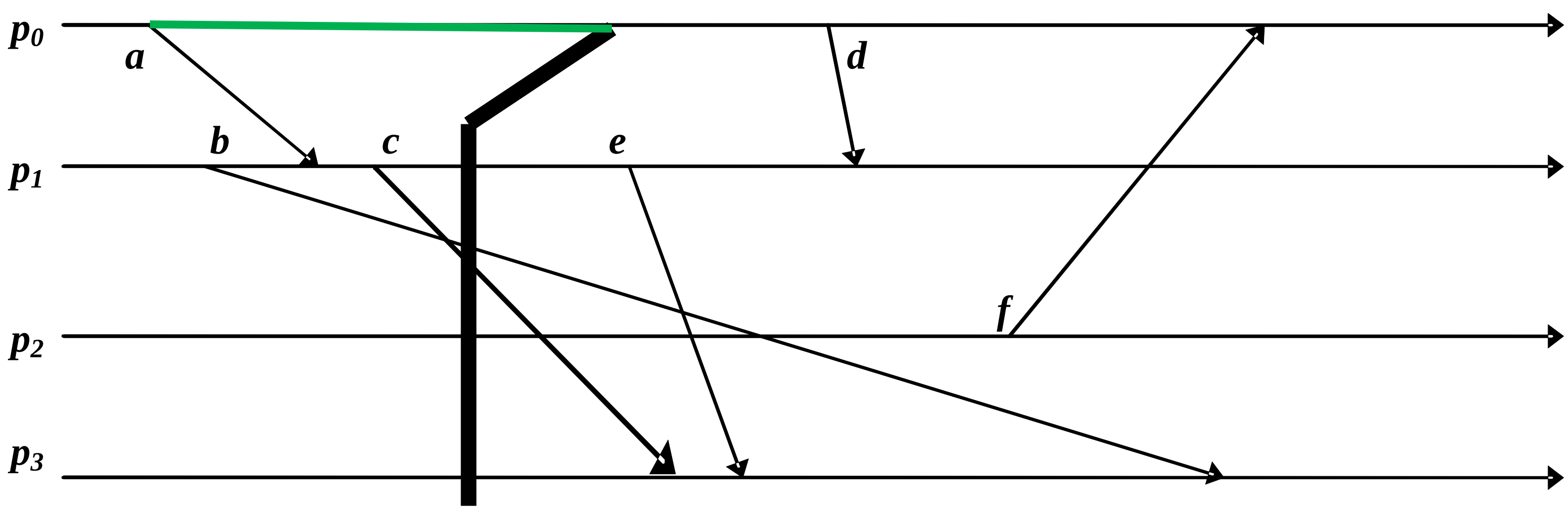
Temporal distortions

- What does "now" mean?



Temporal distortions

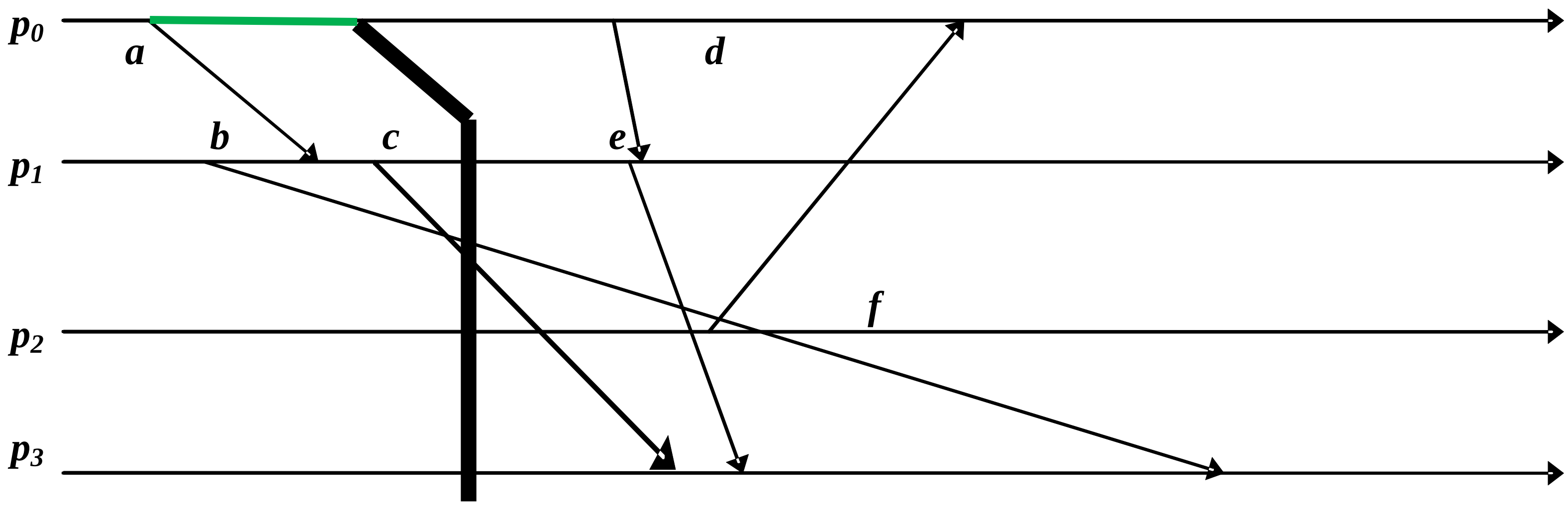
- Timelines can "stretch"…



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

Temporal distortions

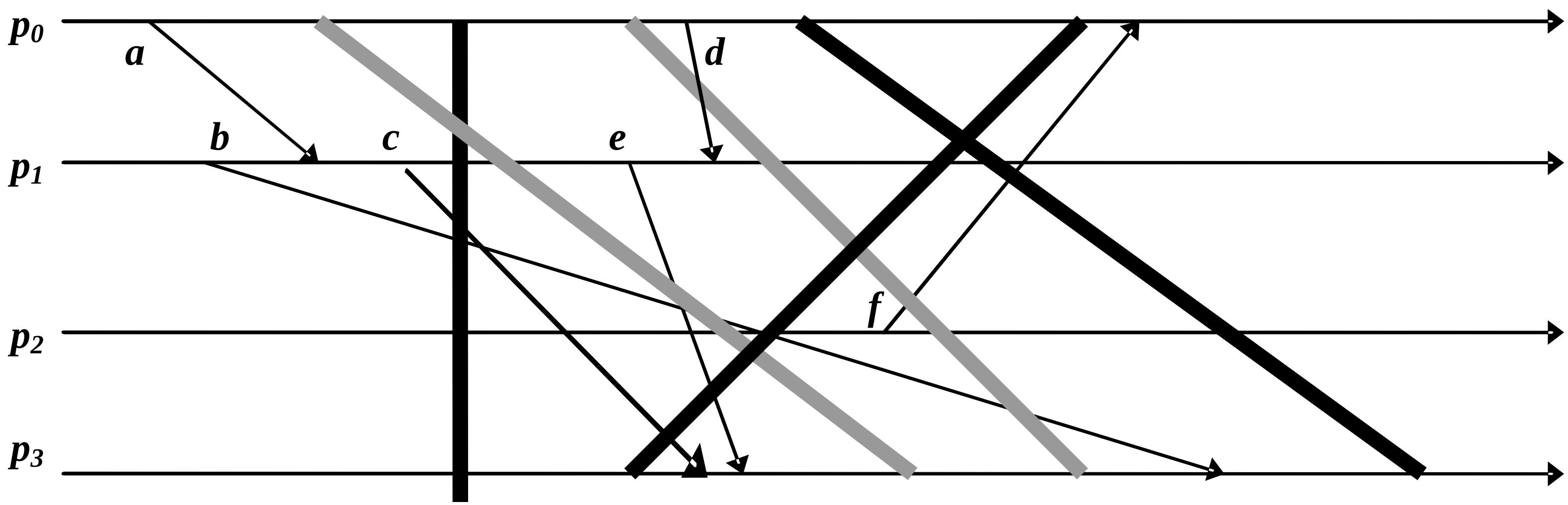
- Timelines can "shrink"



- E.g. something lets a machine speed up

Temporal distortions

- Cuts represent instants of time.



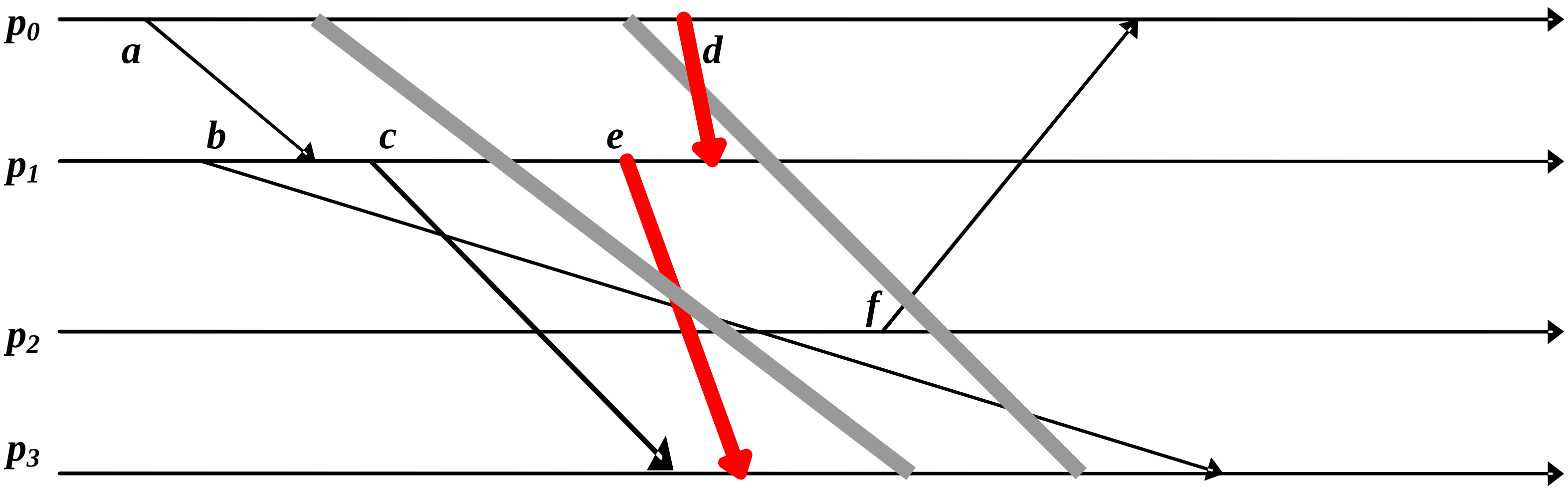
- But not every "cut" makes sense
  - Black cuts could occur but not gray ones.

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

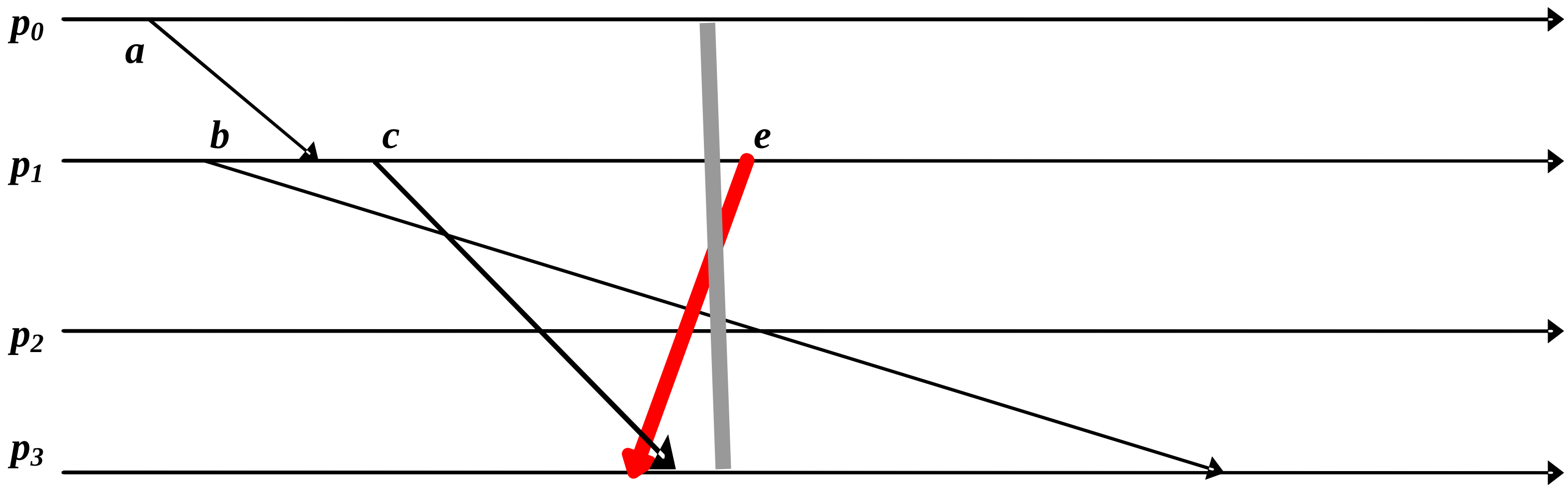
Temporal distortions

- Red messages cross gray cuts "backwards"



Temporal distortions

- Red messages cross gray cuts "backwards"



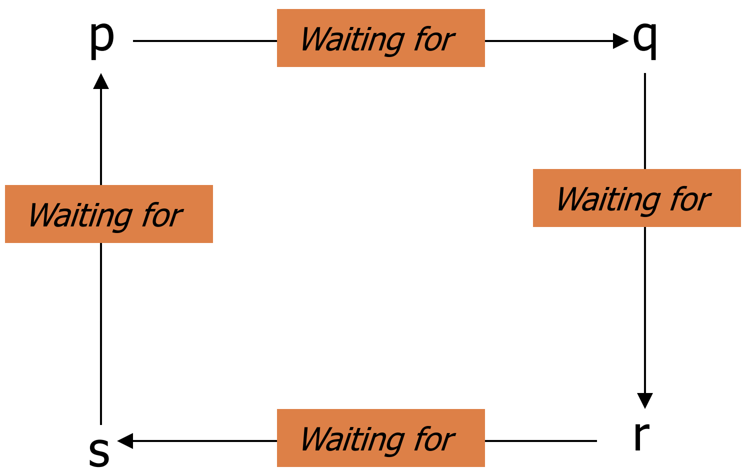
- In a nutshell: the cut includes a message that "was never sent"

Application: Deadlock detection

- 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…

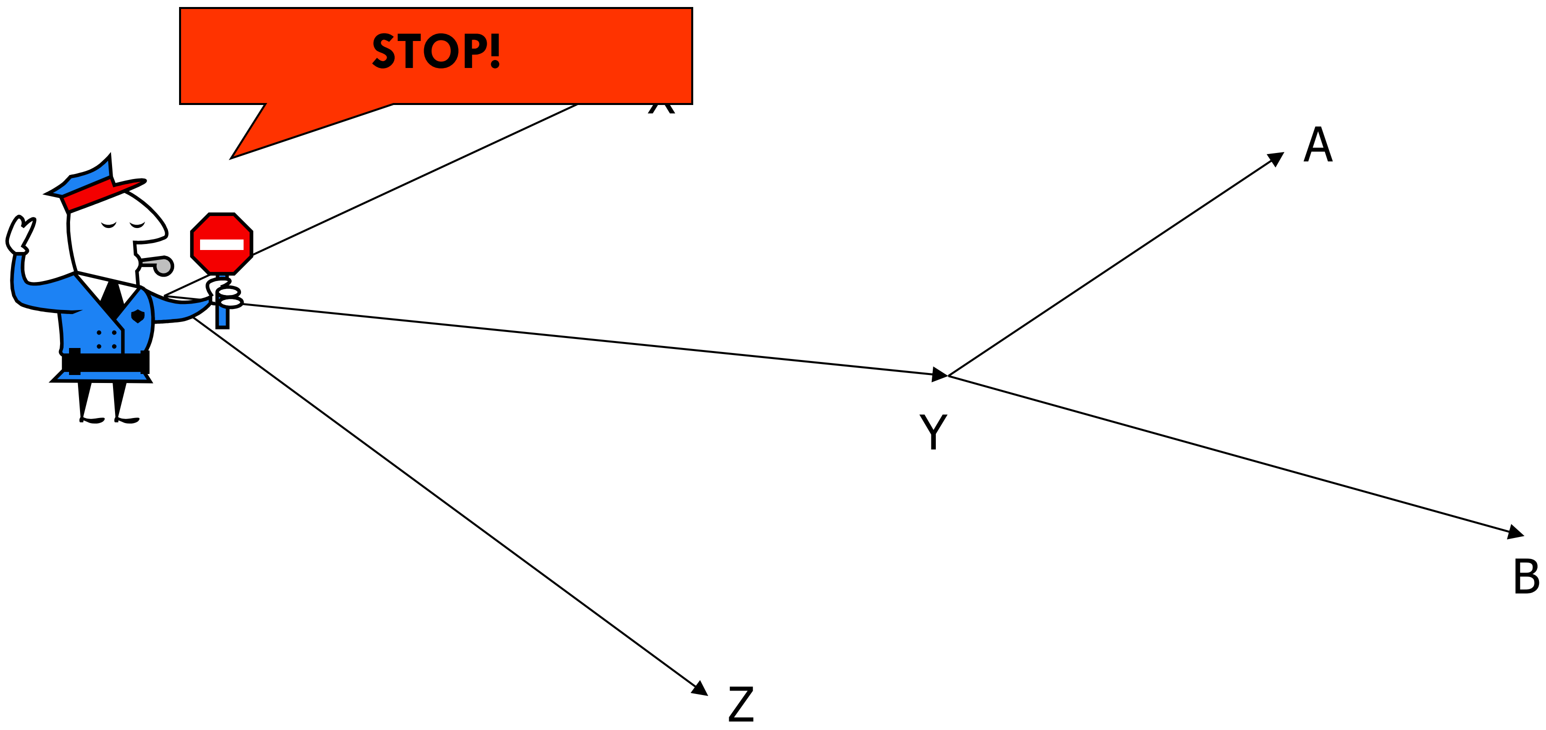


- … but is it a deadlock?

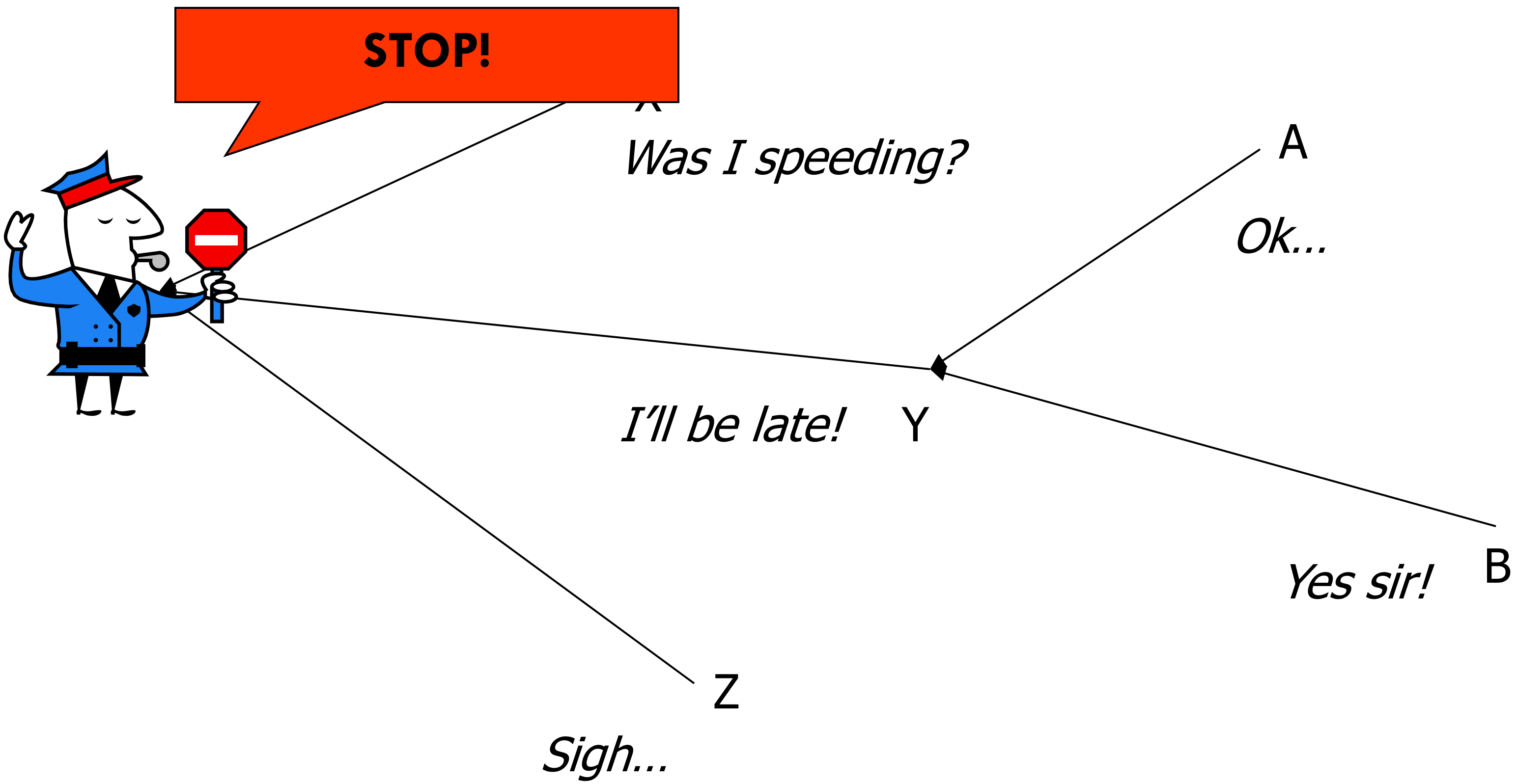
Phantom deadlocks!

- 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.

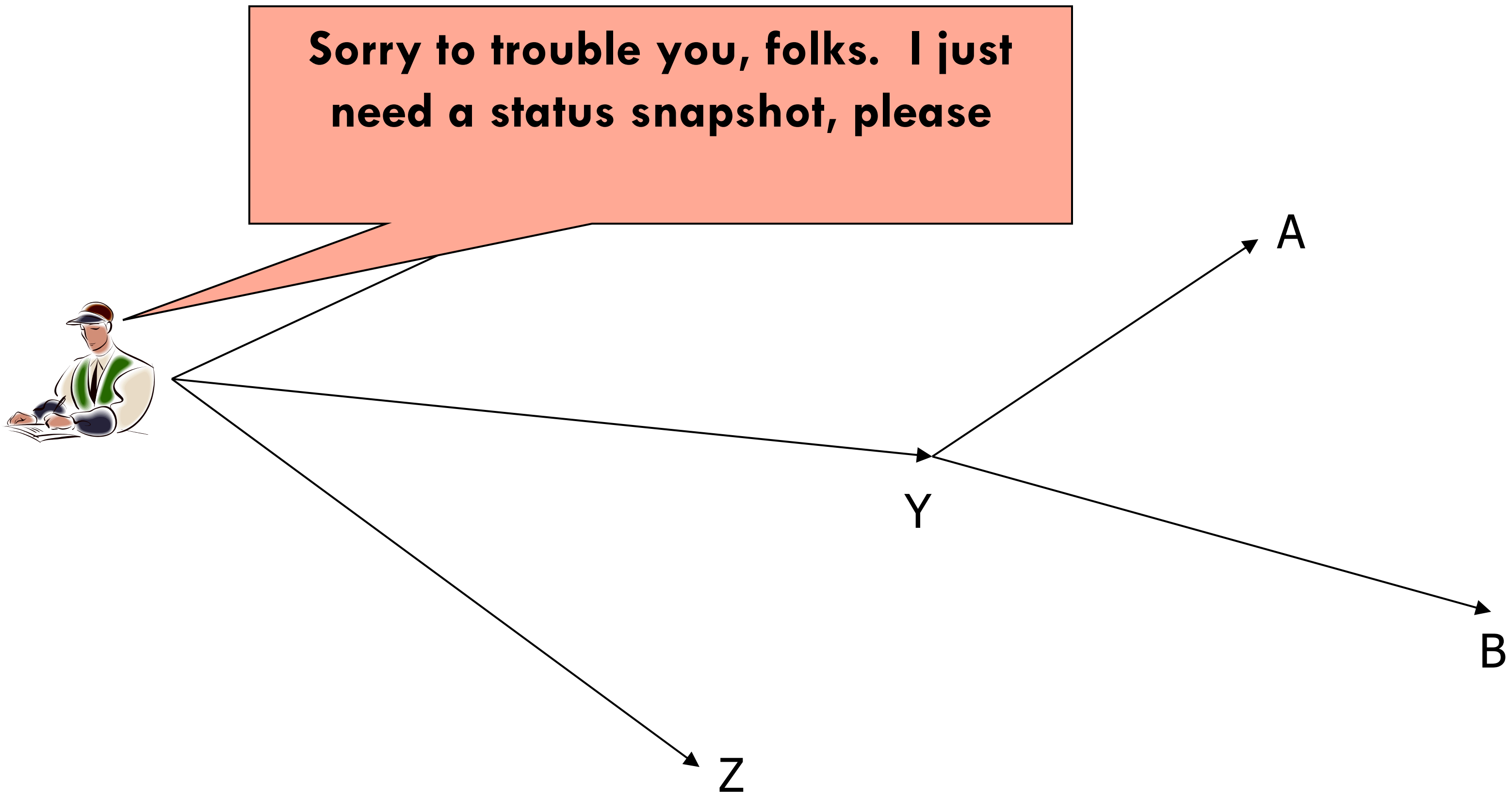
One solution is to “freeze” the system



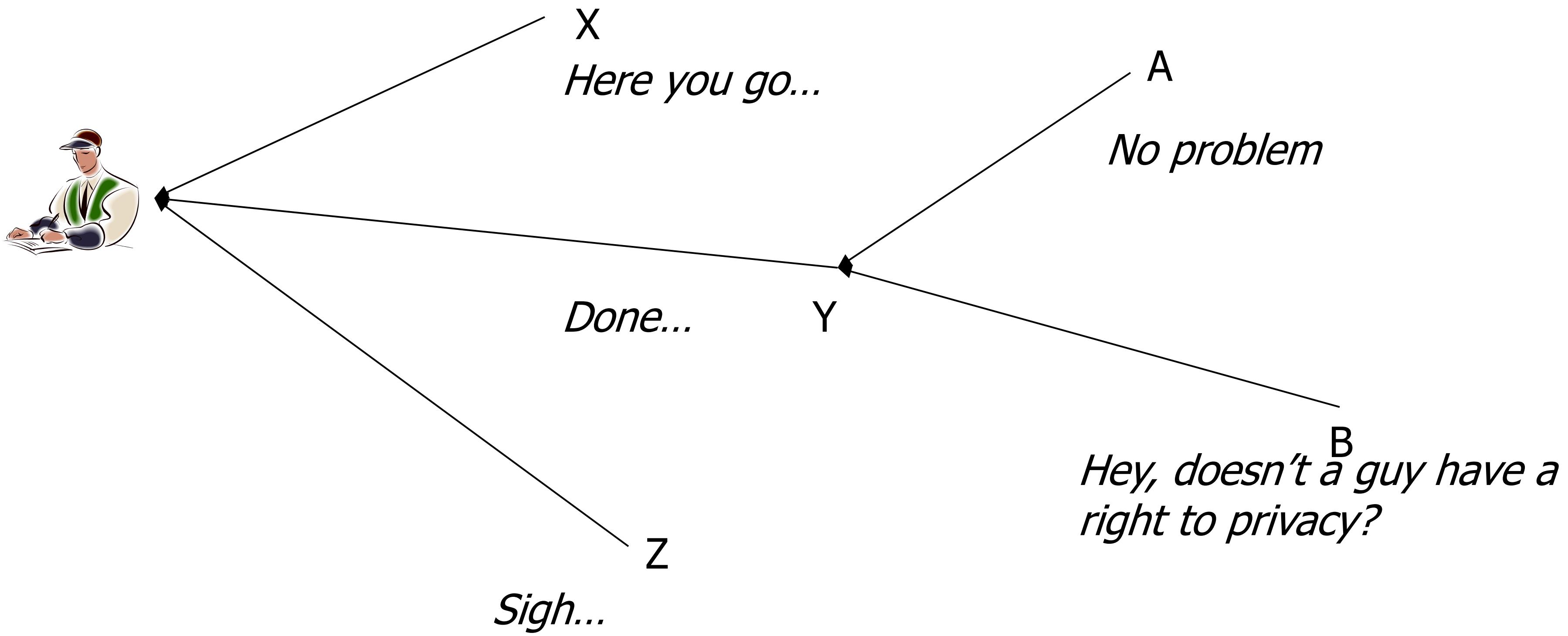
One solution is to “freeze” the system



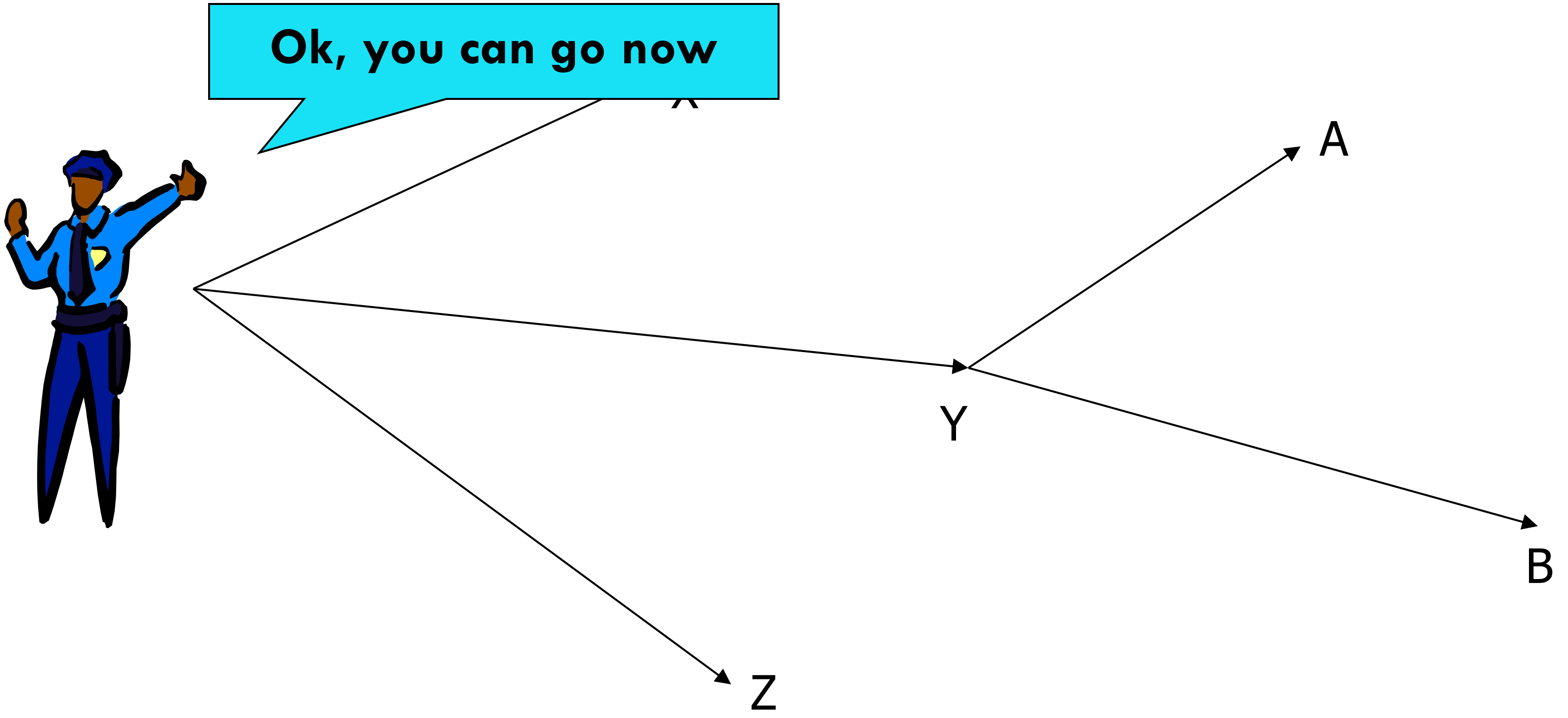
One solution is to “freeze” the system



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

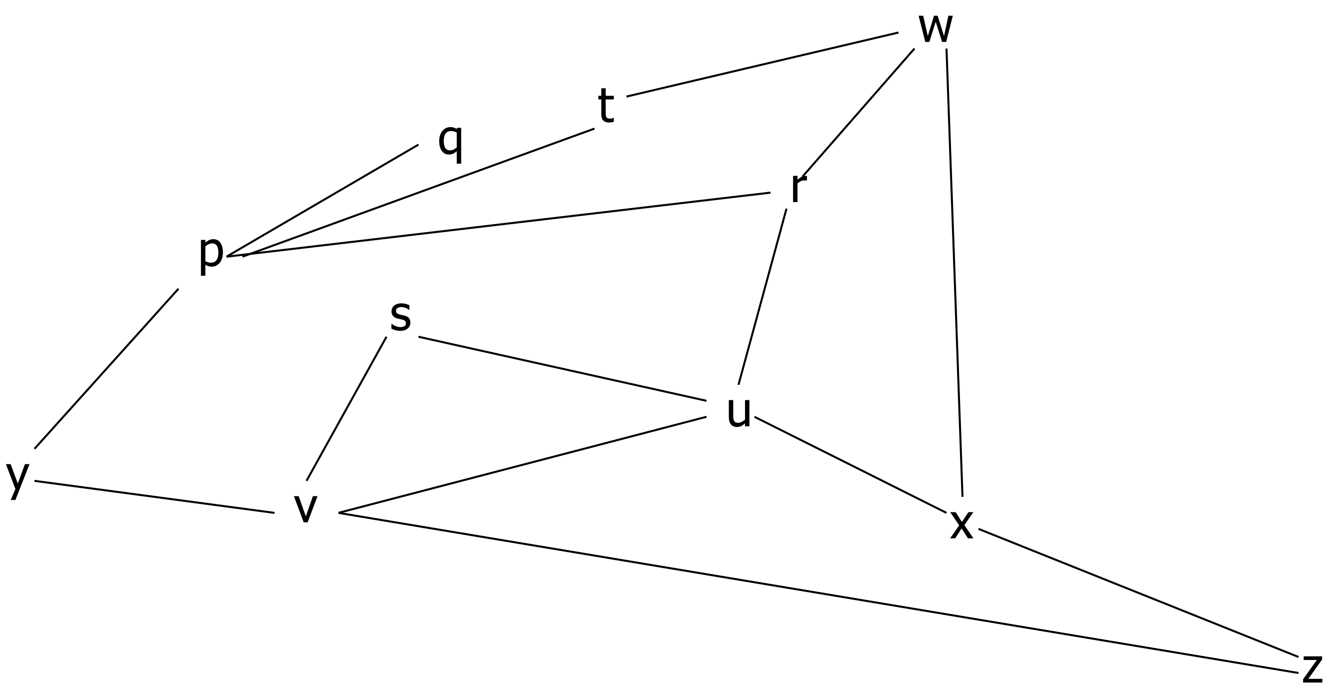
Turn idea into an algorithm

- 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

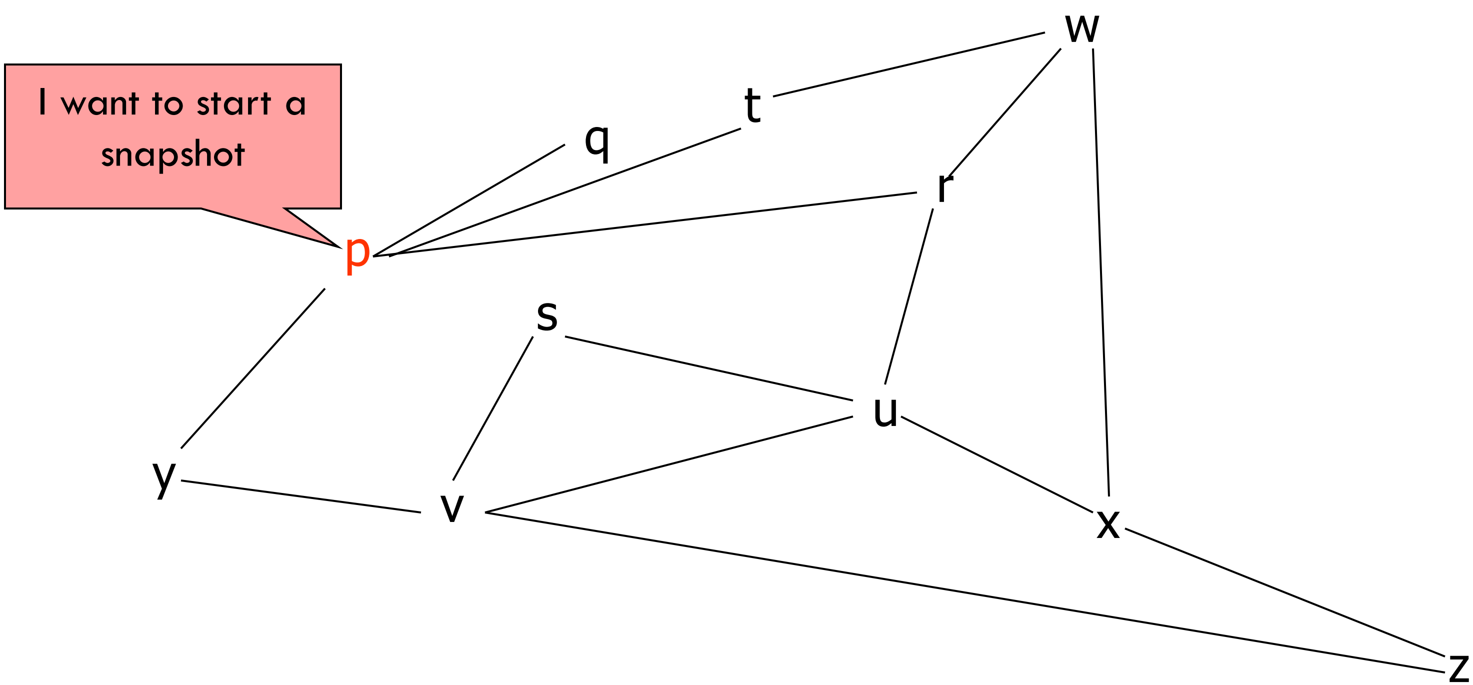
- Outgoing wave of requests… incoming wave of snapshots and channel state
- Snapshot ends up accumulating at the initiator, p_i
- Algorithm doesn't tolerate process failures or message failures.

Chandy/Lamport



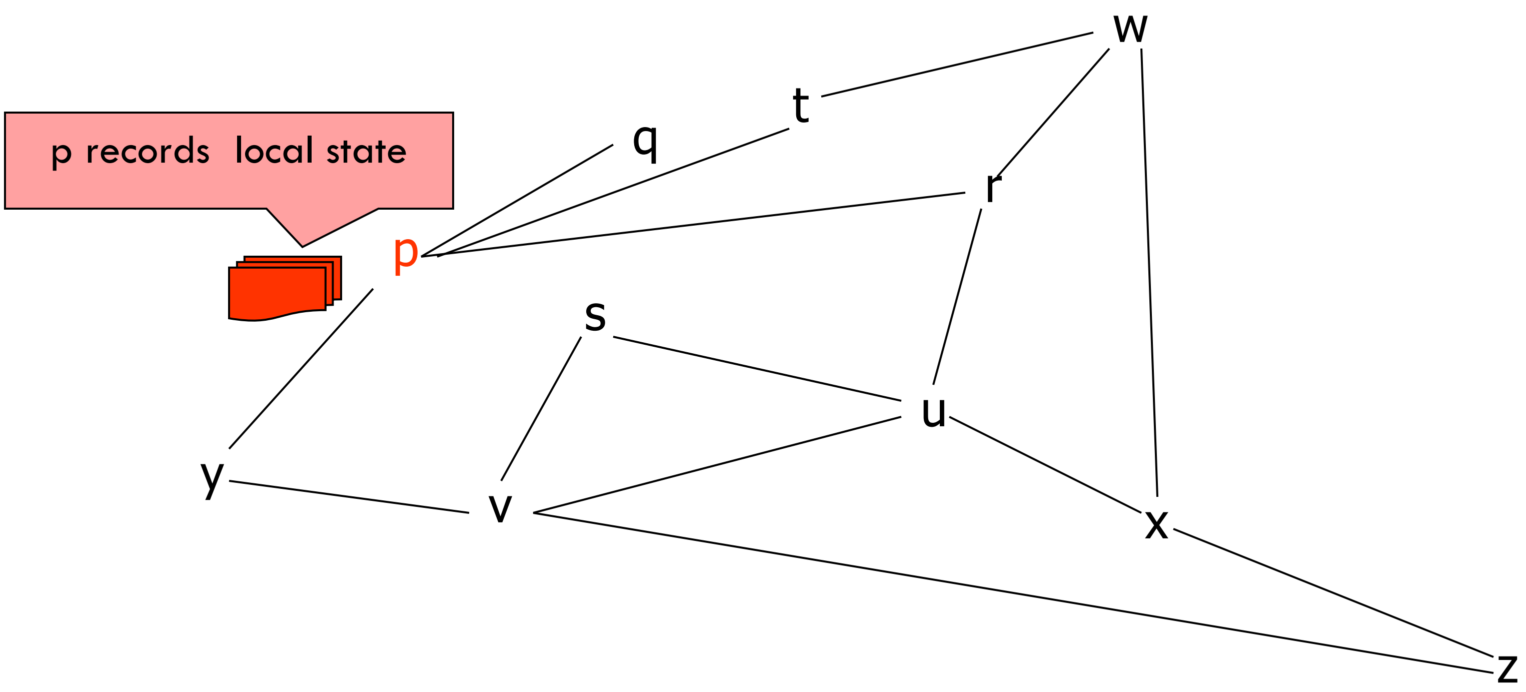
A network

Chandy/Lamport



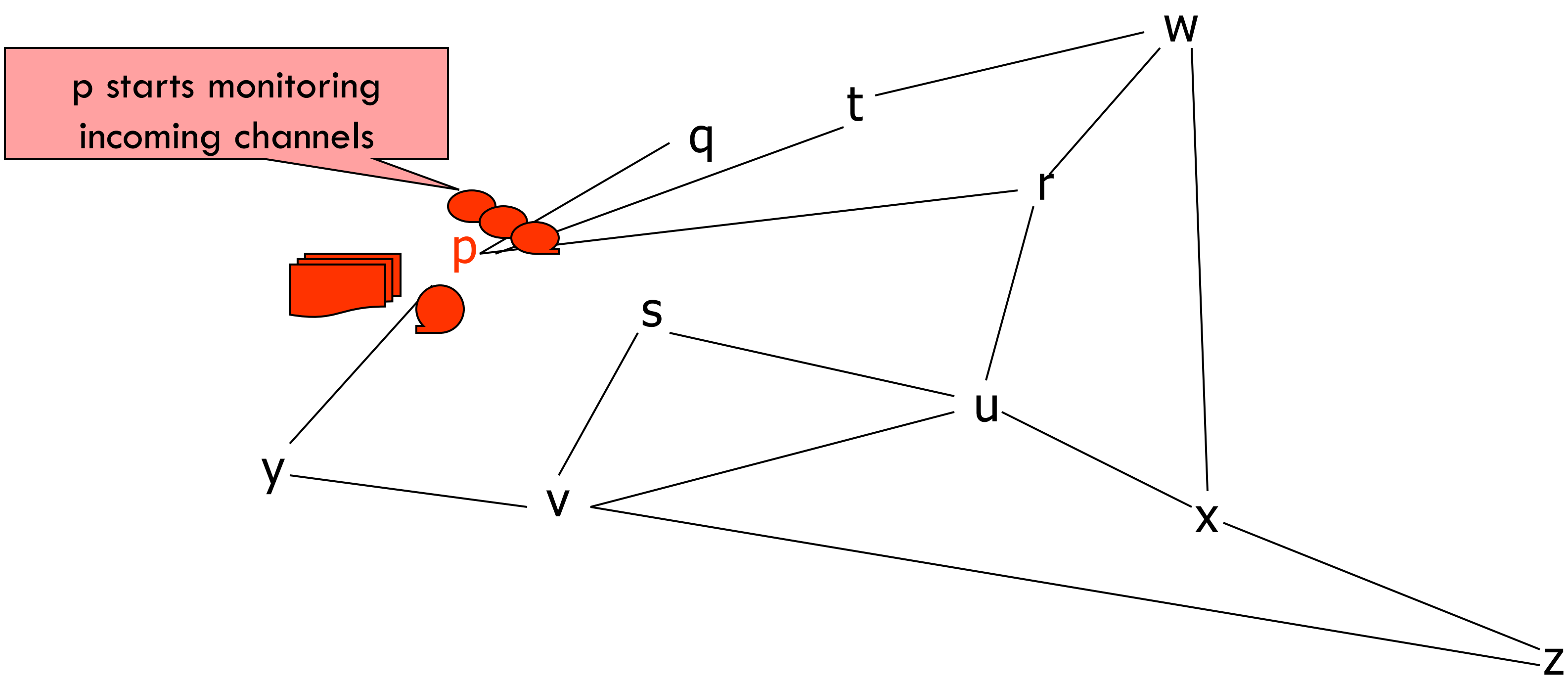
A network

Chandy/Lamport



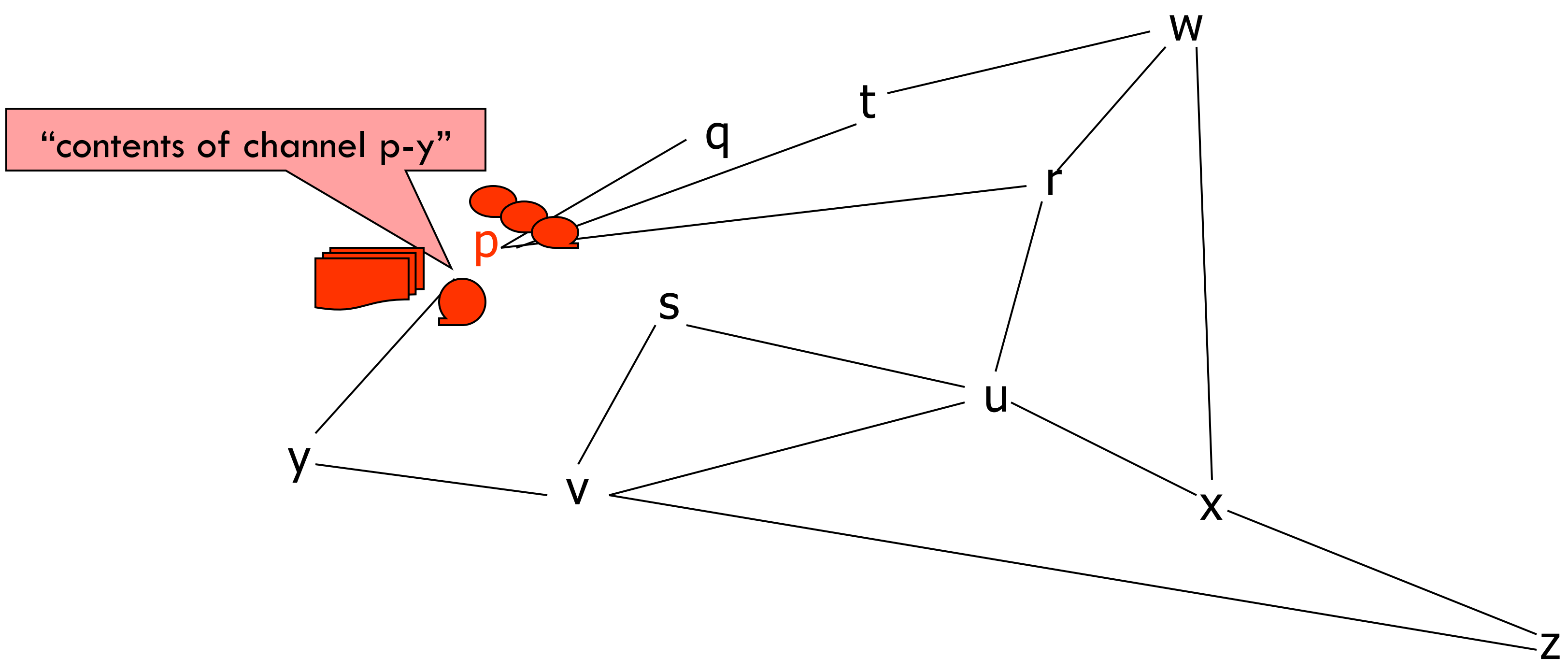
A network

Chandy/Lamport



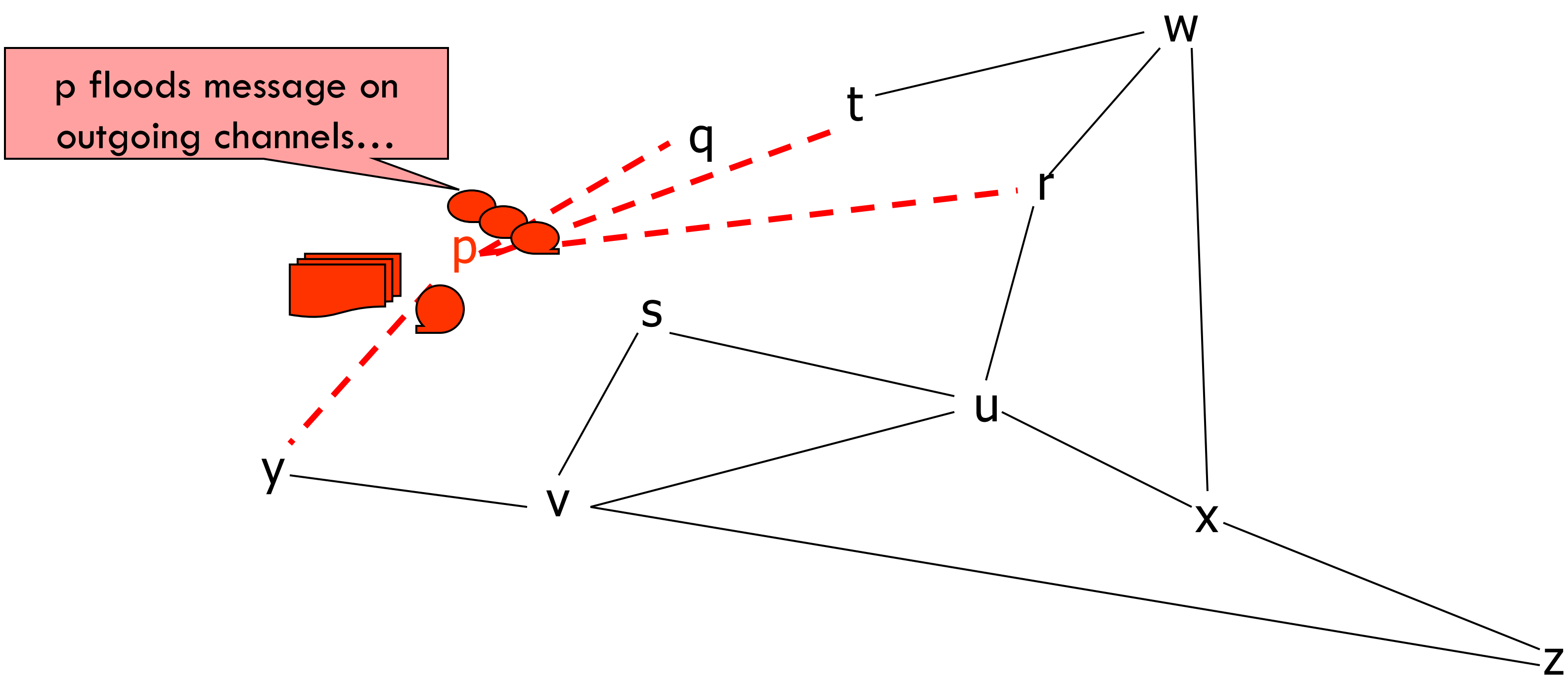
A network

Chandy/Lamport



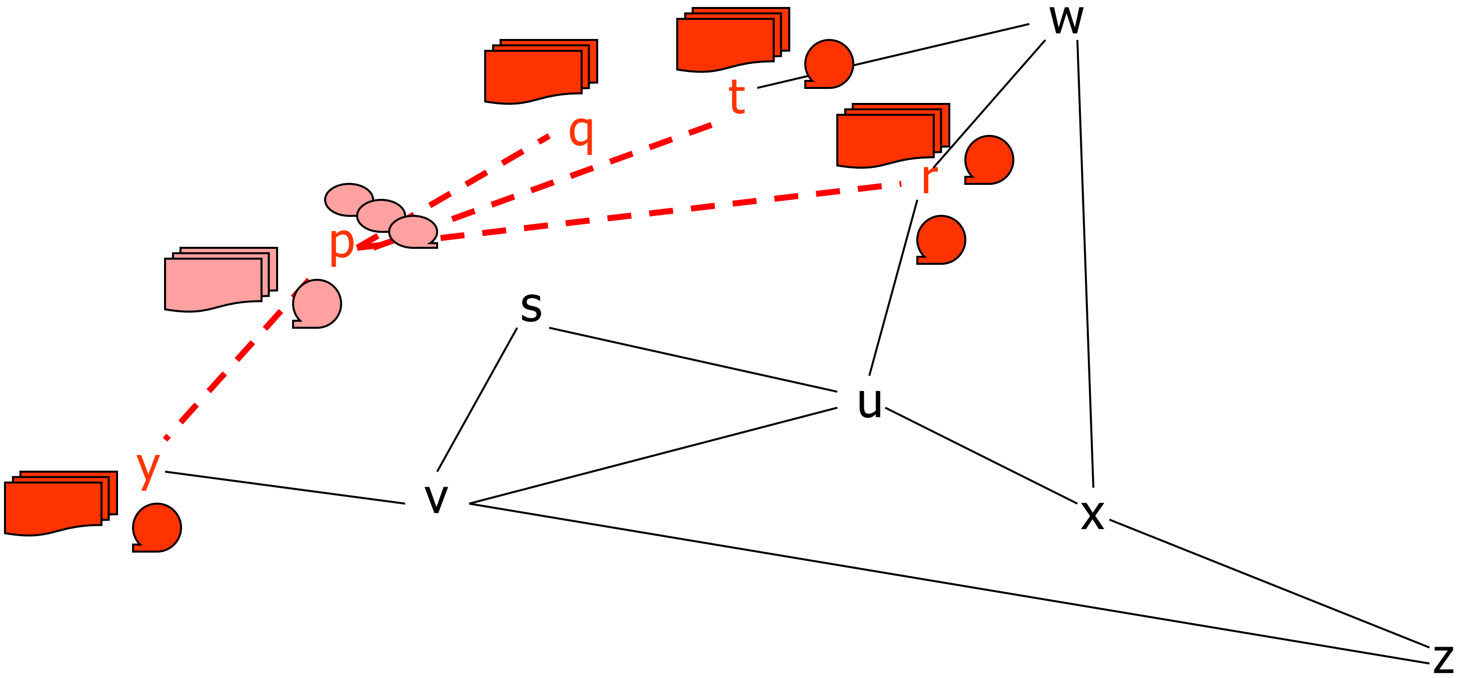
A network

Chandy/Lamport



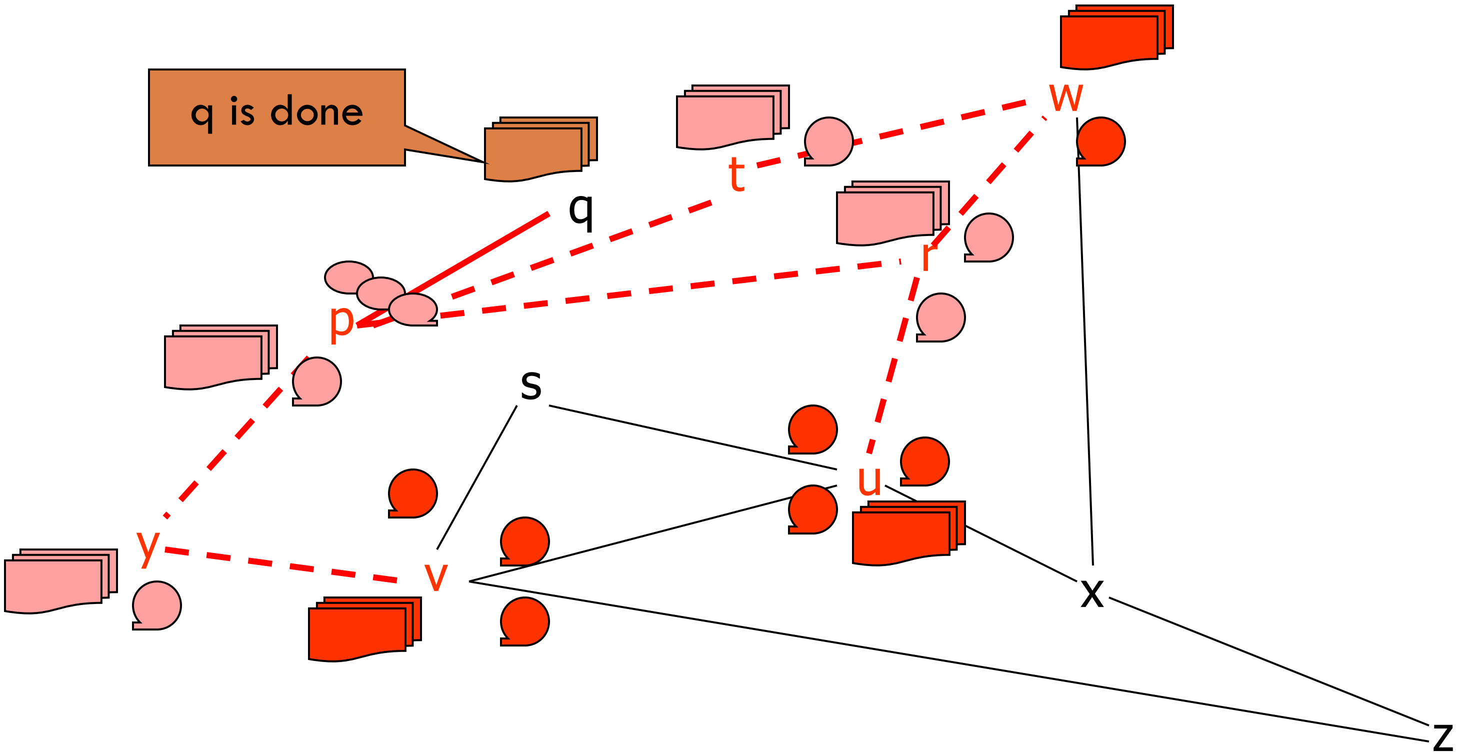
A network

Chandy/Lamport



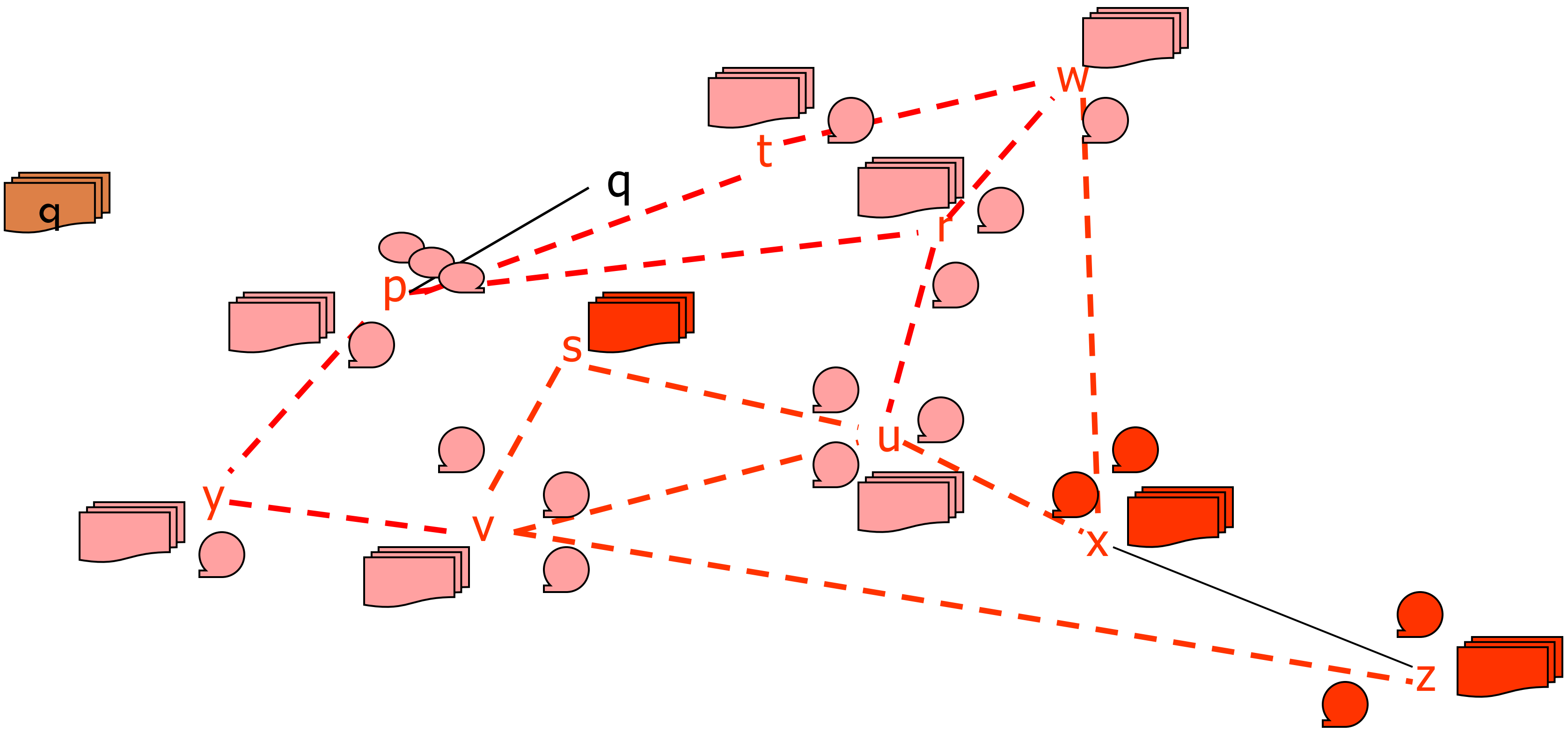
A network

Chandy/Lamport



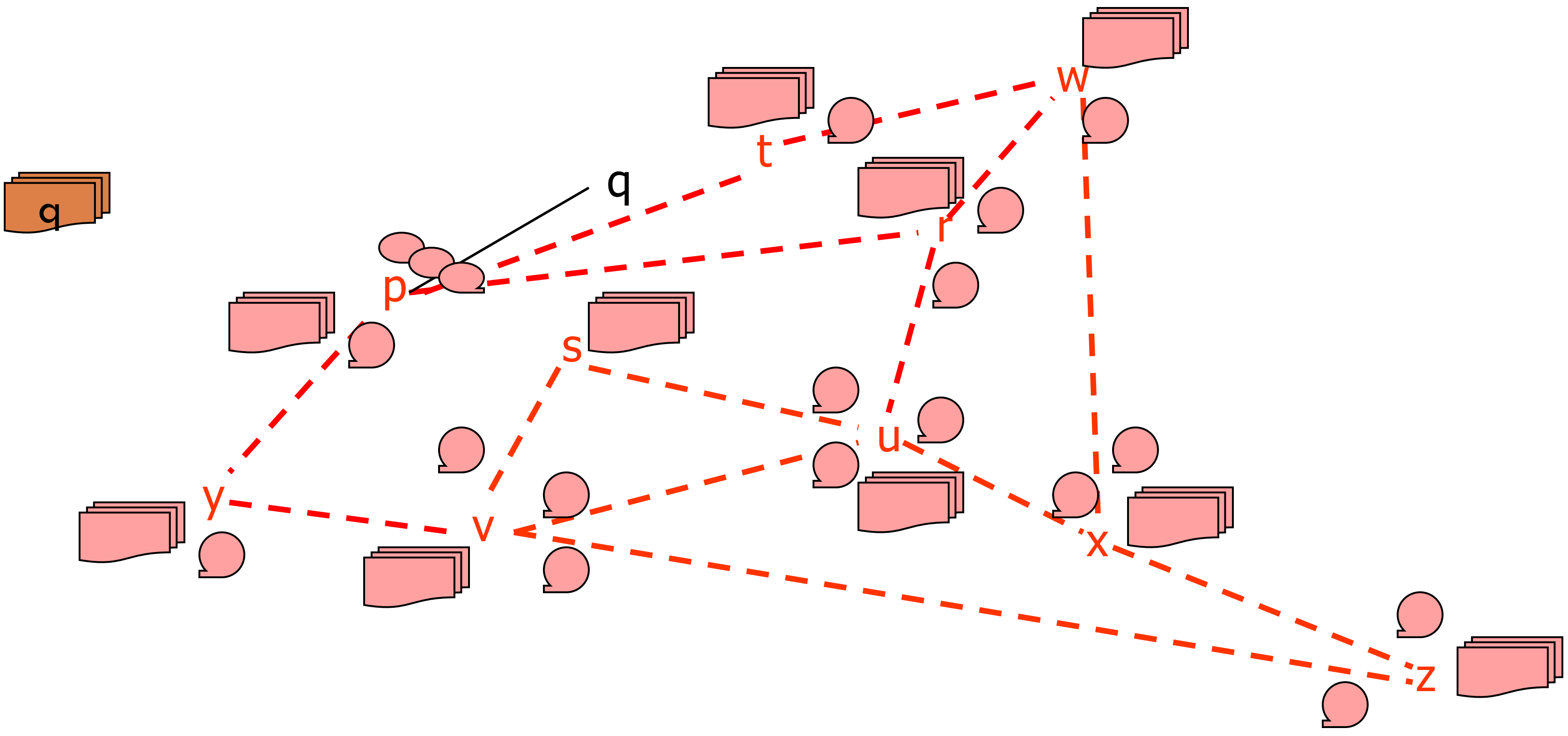
A network

Chandy/Lamport



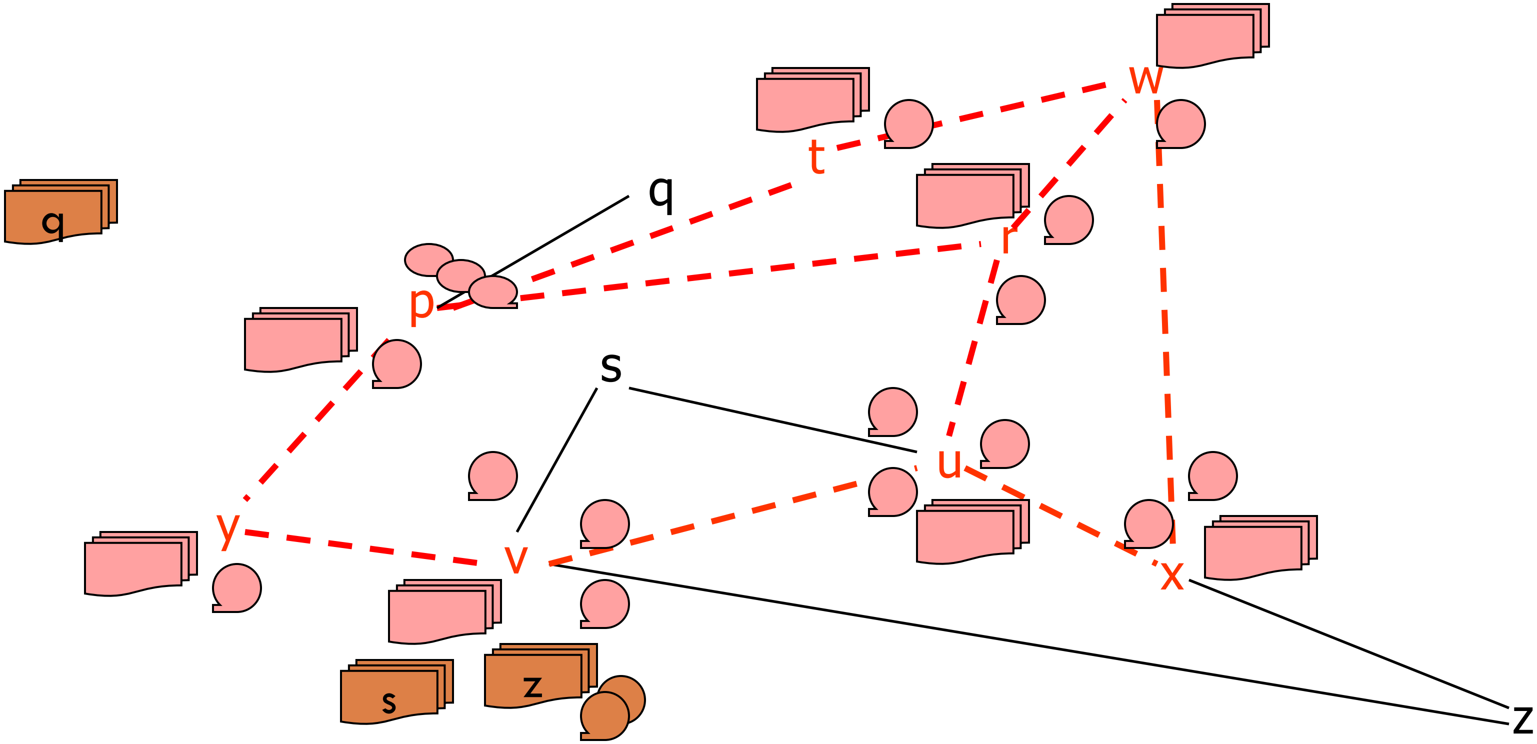
A network

Chandy/Lamport



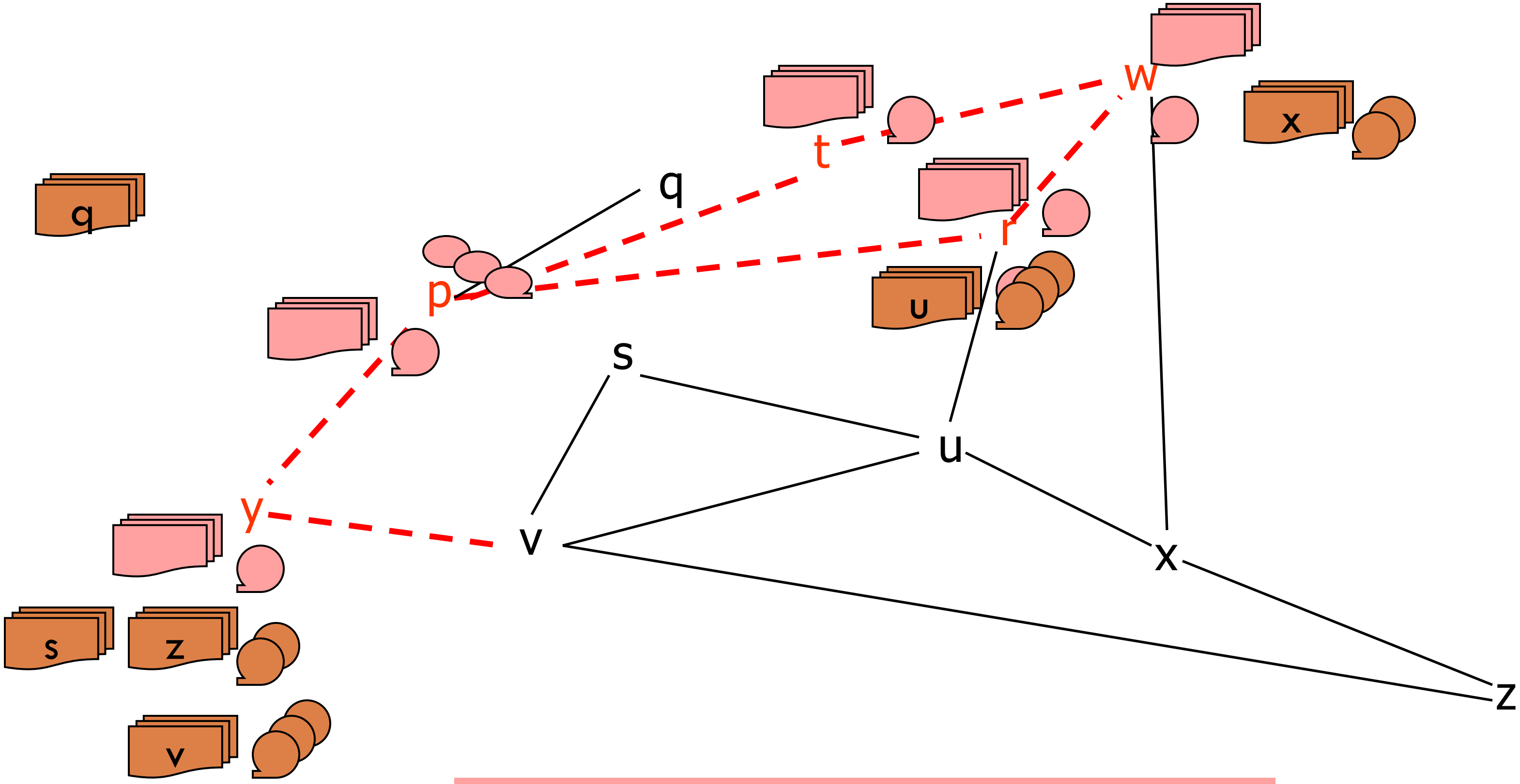
A network

Chandy/Lamport



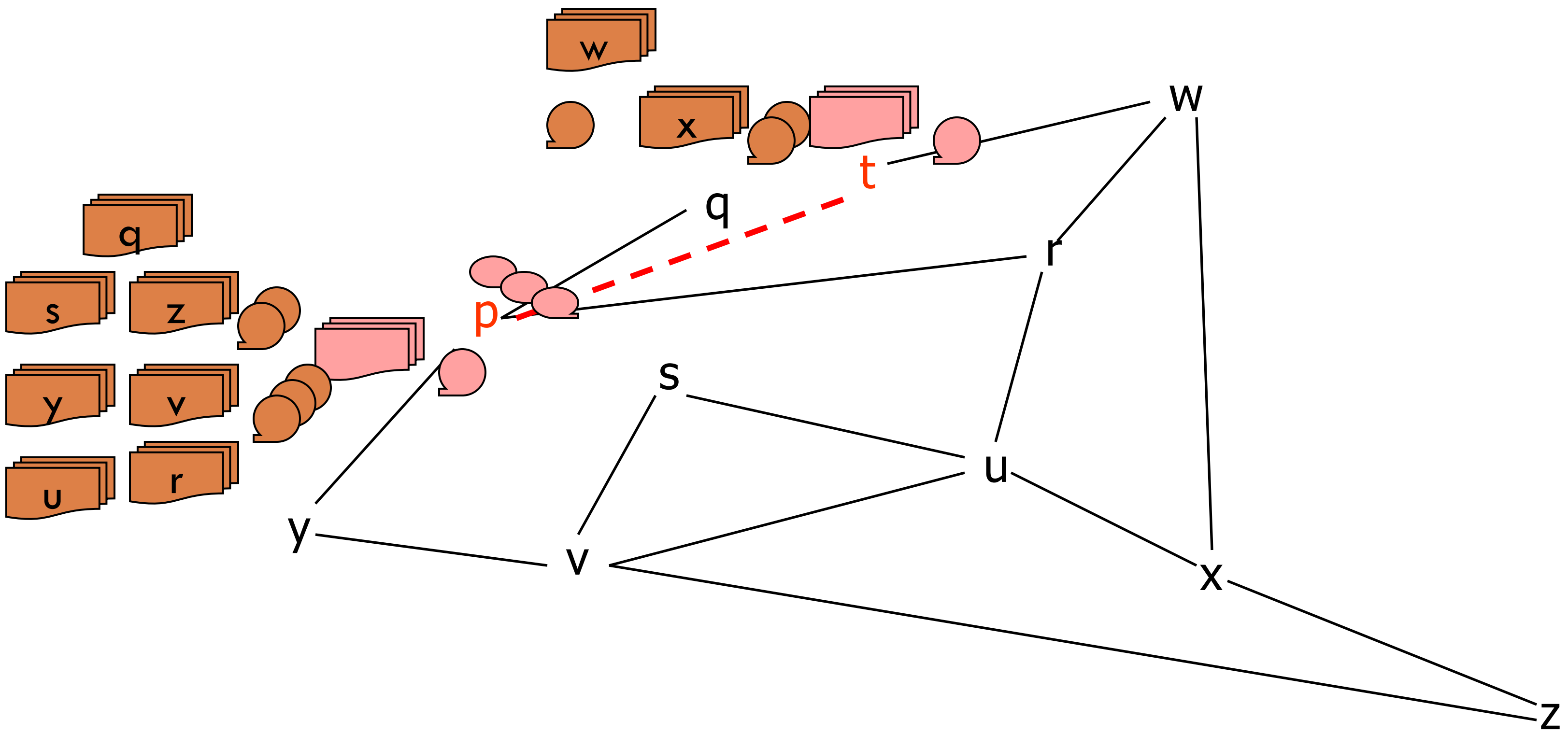
A network

Chandy/Lamport



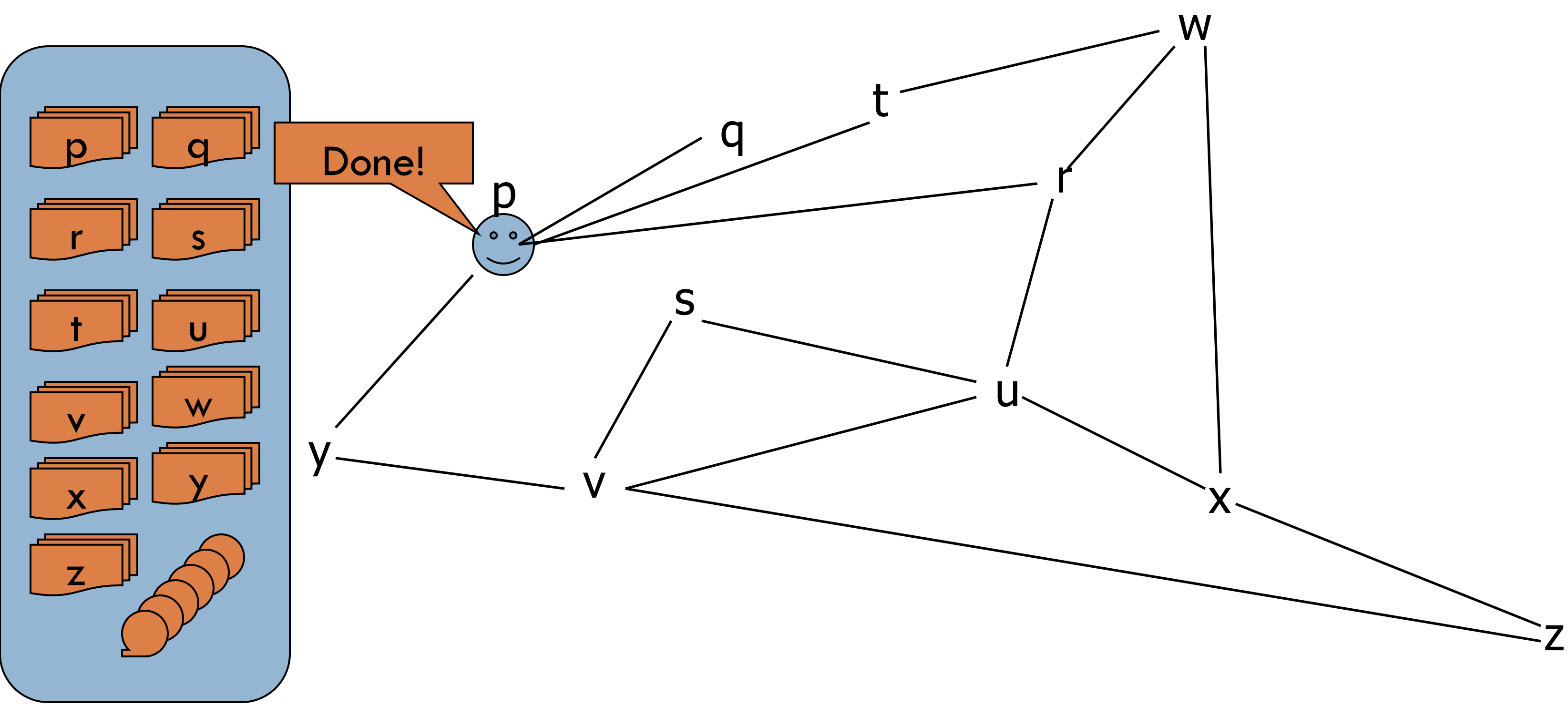
A network

Chandy/Lamport



A network

Chandy/Lamport



A snapshot of a network

Chandy/Lamport “snapshot”

- Once we collect the state snapshots plus the channel contents we have a consistent cut from the system
  - It “could” have occured as a concurrent instant in the system execution (although in fact, it obviously didn’t)
  - Processing such a snapshot requires understanding the state in this form
  - But many algorithms use this *pattern* of messages without necessarily writing down the whole state or logging all the messages in the channels

Relation to vector time?

- In book the connection of consistent cuts to notion of logical time is explored
  - A consistent cut is a snapshot taken at a set of concurrent points in a system trace
  - In effect, all the members of the system concurrently write down their states
  - We can restate Chandy/Lamport to implement it precisely in this manner!
- But out of time today, so we'll leave that for you to read about in Chapter 10 of the text

Conclusions

- By formalizing notion of time we can build tools for thinking about fancier ideas such as consistency of replicated data
- Today we looked more closely at time than at consistency.
  - We introduced idea of consistency to motivate need to look closely at time
  - But didn't tie the logical or vector timestamp ideas back to implementation of replicated data
- Next lectures will make this connection explicit